

R for Excel Users

Julie Lowndes & Allison Horst

2019-11-06

Contents

1	Welcome	7
1.1	Prerequisites	8
2	Overview	9
2.1	Summary (a few sentences)	9
2.2	Objectives (more detailed, bulletpoints?)	9
2.3	Resources	9
2.4	Overview	9
2.5	RStudio Orientation	12
2.6	Deep thought	16
2.7	R Scripts	16
2.8	Don't save the workspace	19
2.9	Deep thought: keep the raw data raw.	19
2.10	Activity 1	19
2.11	Activity 2	19
2.12	Efficiency Tips	19
2.13	Troubleshooting	19
3	readxl	21
3.1	Summary	21
3.2	Objectives	21
3.3	Resources	21
3.4	Lesson	22
3.5	Fun facts ideas:	30
3.6	Interludes (deep thoughts/openscapes)	30
3.7	Activity: Import some invertebrates!	30
3.8	Efficiency Tips	32
4	RMarkdown	33
4.1	Summary (a few sentences)	33
4.2	Objectives (more detailed, bulletpoints?)	33
4.3	Resources	33
4.4	Intro to RMarkdown	33
4.5	Create a Rmd for our fish analysis	37

4.6	Tidy data	37
4.7	Activity	37
4.8	Fun facts (quirky things) - making a note of these wherever possible for interest (little “Did you know?” sections)	38
4.9	Interludes (deep thoughts/openscapes)	38
4.10	Interludes (deep thoughts/openscapes)	38
4.11	Efficiency Tips	38
5	dplyr and Pivot Tables	39
5.1	Summary (a few sentences)	39
5.2	Objectives (more detailed, bulletpoints?)	39
5.3	Resources	39
5.4	Introduction / our analytical plan	39
5.5	What are pivot tables?	40
5.6	dplyr overview	40
5.7	pivot tables: <code>group_by()</code> <code>%>% summarize()</code>	41
5.8	<code>mutate()</code>	43
5.9	<code>mutate()</code> vs <code>summarize()</code>	45
5.10	Deep thoughts	46
5.11	Efficiency Tips	46
6	Dplyr and vlookups	47
6.1	Summary	47
6.2	Objectives	47
6.3	Resources	48
6.4	Lessons	48
6.5	Fun facts	56
6.6	Interludes (deep thoughts/openscapes)	56
6.7	Our Turn Your Turn 1	57
6.8	Our Turn Your Turn 2	57
6.9	Efficiency Tips	57
7	Tidying	59
7.1	Better practices [needs a better name]	59
7.2	Summary (a few sentences)	59
7.3	Objectives (more detailed, bulletpoints?)	59
7.4	Resources	60
7.5	Lessons teaching for each objective..... (objectives, examples)	60
7.6	Fun facts (quirky things) - making a note of these wherever possible for interest (little “Did you know?” sections)	61
7.7	Interludes (deep thoughts/openscapes)	61
7.8	Our Turn Your Turn 2	61
7.9	Efficiency Tips	61
8	Formatting and Sharing	63
8.1	Summary (a few sentences)	63

8.2	Objectives (more detailed, bulletpoints?)	63
8.3	Resources	63
8.4	Introduction / our analytical plan	63
8.5	Formatting tables (15 minutes)	64
8.6	Activity (5 min)	64
8.7	GitHub for Publishing	64
8.8	Create a GitHub user account	65
8.9	Create a repo	65
8.10	Drag and Drop	68
8.11	Visit your website!	68
8.12	Edit the README	68
8.13	Activity (5 min)	69
8.14	recap	69
8.15	Interludes (deep thoughts/openscapes)	69
8.16	Efficiency Tips	69
9	Synthesis	71
9.1	Summary (a few sentences)	71
9.2	Objectives (more detailed, bulletpoints?)	71
9.3	Resources	71
9.4	Lessons teaching for each objective..... (objectives, examples) . .	71
9.5	Fun facts (quirky things) - making a note of these wherever possible for interest (little “Did you know?” sections)	71
9.6	Interludes (deep thoughts/openscapes)	71
9.7	Our Turn Your Turn 1	71
9.8	Our Turn Your Turn 2	71
9.9	Efficiency Tips	71

Chapter 1

Welcome

Excel is a widely used and powerful tool for working with data. As automation, reproducibility, collaboration, and frequent reporting become increasingly expected in data analysis, a good option for Excel users is to extend their workflows with R. Integrating R into data analysis with Excel can bridge the technical gap between collaborators using either software. R enables use of existing tools built for specific tasks and overcomes some limitations that arise when working with large datasets or repeated analyses. This course is for Excel users who want to add or integrate R and RStudio into their existing data analysis toolkit. Participants will get hands-on experience working with data across R, Excel, and Google Sheets, focusing on: data import and export, basic wrangling, visualization, and reporting with RMarkdown. Throughout, we will emphasize conventions and best practices for working reproducibly and collaboratively with data, including naming conventions, documentation, organization, all while “keeping the raw data raw”. Whether you are working in Excel and want to get started in R, already working in R and want tools for working more seamlessly with collaborators who use Excel, or whether you are new to data analysis entirely, this is the course for you!

If you answer yes to these questions, this course is for you!

- Are you an Excel user who wants to expand your data analysis toolset with R?
- Do you want to bridge analyses between Excel and R, whether working independently or to more easily collaborate with others who use Excel or R?
- Are you new to data analysis, and looking for a good place to get started?

1.1 Prerequisites

Before the training, please make sure you have done the following:

1. Download and install **up-to-date versions** of:
 - R: <https://cloud.r-project.org>
 - RStudio: <http://www.rstudio.com/download>
2. Install the Tidyverse
3. Get comfortable: if you're not in a physical workshop, be set up with two screens if possible. You will be following along in RStudio on your own computer while also watching a virtual training or following this tutorial on your own.

Chapter 2

Overview

2.1 Summary (a few sentences)

2.2 Objectives (more detailed, bulletpoints?)

- working with data that are not your own.

2.3 Resources

R is not only a language, it is an active community of developers, users, and educators (often these traits are in each person). This workshop and book based on many excellent materials created by other members in the R community, who share their work freely to help others learn. Using community materials is how WE learned R, and each chapter of the book will have Resources listed for further reading into the topics we discuss. And, when there is no better way to explain something (ahem Jenny Bryan), we will quote or reference that work directly.

- What They Forgot to Teach You About R — Jenny Bryan & Jim Hester
- Stat545 — Jenny Bryan & Stat545 TAs
- Where do Things Live in R? REX Analytics
-

2.4 Overview

Welcome.

This workshop you will learn hands-on how to begin to interoperate between Excel and R.

A main theme throughout is to produce analyses people can understand and build from — including Future You. Not so brittle/sensitive to minor changes.

We will learn and reinforce X main things all at the same time: coding with best practices (R/RStudio), how this relates to operations in Excel, Z. This training will teach these all together to reinforce skills and best practices, and get you comfortable with a workflow that you can use in your own projects.

2.4.1 What to expect

This is going to be a fun workshop.

The plan is to expose you to X that you can have confidence using in your work. You'll be working hands-on and doing the same things on your own computer as we do live on up on the screen. We're going to go through a lot in these two days and it's less important that you remember it all. More importantly, you'll have experience with it and confidence that you can do it. The main thing to take away is that there *are* good ways to work between R and Excel; we will teach you to expect that so you can find what you need and use it! A theme throughout is that tools exist and are being developed by real, and extraordinarily nice, people to meet you where you are and help you do what you need to do. If you expect and appreciate that, you will be more efficient in doing your awesome science.

You are all welcome here, please be respectful of one another. You are encouraged to help each other.

Everyone in this workshop is coming from a different place with different experiences and expectations. But everyone will learn something new here, because there is so much innovation in the data science world. Instructors and helpers learn something new every time, from each other and from your questions. If you are already familiar with some of this material, focus on how we teach, and how you might teach it to others. Use these workshop materials not only as a reference in the future but also for talking points so you can communicate the importance of these tools to your communities. A big part of this training is not only for you to learn these skills, but for you to also teach others and increase the value and practice of open data science in science as a whole.

2.4.2 What you'll learn

TODO: dev

- Motivation is to bridge and/or get out of excel
- We're not going to replicate all of your fancy things in R,

- We use Excel to look at data that we’re reading into R
- Spreadsheets are great; blend data entry with analyses and we’re going to try to help you think about them a bit more distinctively.
- Most important collaborator is future you, and future us

An important theme for this workshop is being deliberate about your analyses and setting things up in a way that will make your analytical life better downstream in the current task, and better when Future You or Future Us revisit it in the future (i.e. avoiding: what happens next? What does this name mean?)

This graphic by Hadley Wickham and Garrett Grolemund in their book R for Data Science is simple but incredibly powerful:

You may not have ever thought about analysis in such discrete steps: I certainly hadn’t before seeing this. That is partly because in Excel, it can be easy to blend these steps together. We are going to keep them separate, and talk about why. The first step is Import: and implicit in this as a first step is that the data is stored elsewhere and is not manipulated directly, which **keeps the raw data raw**.

We will be focusing on:

- **Import:** `readr`, `readxl` to read raw data stored in CSV or Excel files directly into R
- **Tidy:** `tidyr` to (re)organize rows of data into unique values
- **Transform:** `dplyr` to “wrangle” data based on subsetting by rows or columns, sorting and joining
- **Visualize:** `ggplot2` static plots, using grammar of graphics principles
- **Communicate**
 - `writexl` to export intermediate and final data
 - GitHub File Upload and Issues for online publishing and collaboration

2.4.3 Emphasizing collaboration

TODO: rewrite/update (from OHI book):

Collaborating efficiently has historically been really hard to do. It’s only been the last 20 years or so that we’ve moved beyond mailing things with the postal service. Being able to email and get feedback on files through track changes was a huge step forward, but it comes with a lot of bookkeeping and reproducibility issues (did I send that report based on `analysis_final_final.xls` or `analysis_final_usethisone.xls`?). But now, open tools make it much easier to collaborate.

Working with collaborators in mind is critical for reproducibility. And, your most important collaborator is Future You. This training will introduce best

practices using open tools, so that collaboration will become second nature to you!

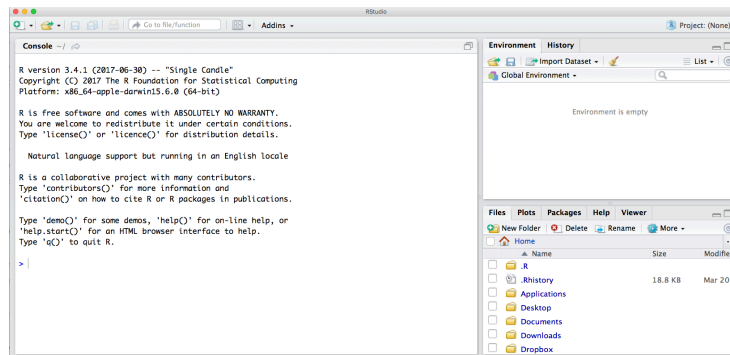
2.4.4 By the end of the course...

By the end of this course you'll produce this report that you can reproduce, which means... Introduce the problem we will solve. Eg: (just an idea maybe time-series is not a great idea) SMALL PROBLEM. (4 mins) Show data files, We will discuss our analysis plan (only enough to motivate!) Create a report, that looks great.

2.5 RStudio Orientation

Open RStudio for the first time.

Launch RStudio/R.



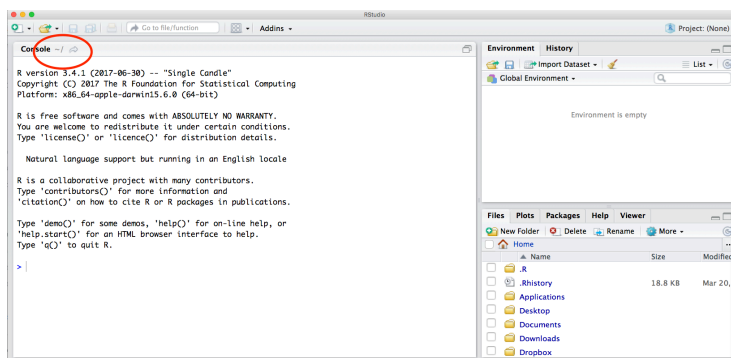
Notice the default panes:

- Console (entire left)
- Environment/History (tabbed in upper right)
- Files/Plots/Packages/Help (tabbed in lower right)

FYI: you can change the default location of the panes, among many other things: Customizing RStudio.

An important first question: **where are we?**

If you've have opened RStudio for the first time, you'll be in your Home directory. This is noted by the ~/ at the top of the console. You can see too that the Files pane in the lower right shows what is in the Home directory where you are. You can navigate around within that Files pane and explore, but note that you won't change where you are: even as you click through you'll still be Home: ~/.



2.5.1 RStudio Projects

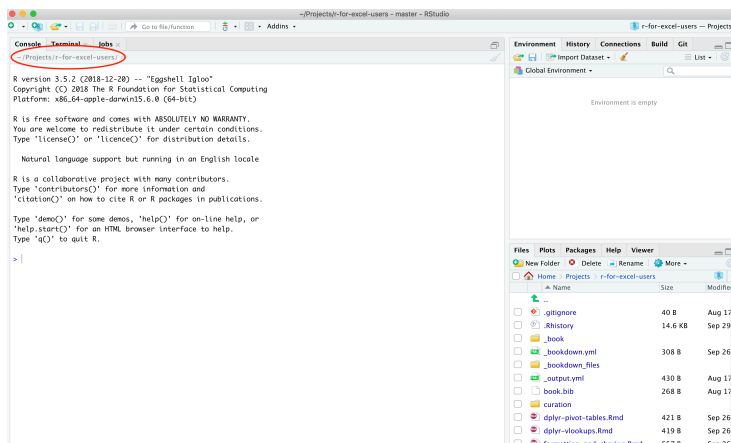
Create a new Project called ‘r-for-excel-users’. File > New Project... > New Directory > New Project. Give your Project a name browse to a place to keep it. And then click to Create Project!

What is a Project? It is a way to organize all of the relevant things you need for an analysis in the same place. This means data, code, figures, notes, etc.

Why does this matter? Keeping everything you need for your analysis together makes it less brittle and more portable — across people, time, and computers.

Working directory = no file path/broken path issues. Notice that a folder now appears wherever you saved this project with the same name, and it contains a .Rproj file.

Now that we have our Project, here is an important question: where are we? Now we are in our Project. Everything we do will by default be saved here so we can be nice and organized.



2.5.2 R Console

Watch me work in the Console.

I can do math:

```
52*40  
365/12
```

TODO: refine

But like Excel, the power comes not from doing small operations by hand (like $8*22.3$), it's by being able to operate on whole suites of numbers and datasets. In Excel, data are stored in the spreadsheet. In R, they are stored in dataframes, and named as variables.

R stores data in variables, and you give them names. This is a big difference with Excel, where you usually identify data by its location on the grid, like `$A1:D$20`. (You can do this with Excel by naming ranges of cells, but most people don't do this.)

Data can be a variety of formats, like numeric and text.

Let's have a look at some data in R. R has several built-in data sets that we can look at and work with.

One of these datasets is called `mtcars`. If I write this in the Console, it will print the data in the console.

```
mtcars
```

I can also use RStudio's Viewer to see this in a more familiar-looking format:

```
View(mtcars)
```

This opens the fourth pane of the RStudio IDE; when you work in R you will have all four panes open so this will become a very comforting setup for you.

The basic R data structure is a vector. You can think of a vector like a column in an Excel spreadsheet with the limitation that all the data in that vector must be of the same type. If it is a character vector, every element must be a character; if it is a logical vector, every element must be TRUE or FALSE; if it's numeric you can trust that every element is a number. There's no such constraint in Excel: you might have a column which has a bunch of numbers, but then some explanatory text intermingled with the numbers. This isn't allowed in R. - https://blog.shotwell.ca/posts/r_for_excel_users/

In the Viewer I can do things like filter or sort. This does not do anything to the actual data, it just changes how you are viewing the data. So even as I explore it, I am not editing or manipulating the data.

Like Excel, some of the biggest power in R is that there are built-in functions that you can use in your analyses (and, as we'll see, R users can easily create and share functions, and it is this open source developer and contributor community that makes R so awesome).

So let's look into some of these functions. In Excel, there is a "SUM" function to calculate a total. Let's expect that there is the same in R. I will type this into the Console:

```
?sum
```

A few important things to note:

1. R is case-sensitive. So "sum" is a completely different thing to "Sum" or "SUM". And this is true for the names of functions, data sets, variable names, and data itself ("blue" vs "Blue").
2. RStudio has an autocomplete feature that can help you find the function you're looking for. In many cases it pops up as you type, but you can always type the tab key (above your caps lock key) to prompt the autocomplete. And, bonus: this feature can help you with the case-sensitivity mentioned above: If I start typing "?SU" and press tab, it will show me all options starting with those two letters, regardless of capitalization (although it will start with the capital S options).

OK but what does typing `?sum` actually *do*?

When I press enter/return, it will open up a help page in the bottom right pane. Help pages vary in detail I find some easier to digest than others. But they all have the same structure, which is helpful to know. The help page tells the name of the package in the top left, and broken down into sections:

- Description: An extended description of what the function does.
- Usage: The arguments of the function and their default values.
- Arguments: An explanation of the data each argument is expecting.
- Details: Any important details to be aware of.
- Value: The data the function returns.
- See Also: Any related functions you might find useful.
- Examples: Some examples for how to use the function.

When I look at a help page, I start with the description, which might be too in-the-weeds for the level of understanding I need at the offset. For the `sum` page, it is pretty straight-forward and lets me know that yup, this is the function I want.

I next look at the usage and arguments, which give me a more concrete view into what the function does. This syntax looks a bit cryptic but what it means is that you use it by writing `sum`, and then passing whatever you want to it in terms of data: that is what the "..." means. And the `"na.rm=FALSE"` means that by default, it will not remove NAs (I read this as: "remove NAs? FALSE!")

Then, I usually scroll down to the bottom to the examples. This is where I can actually see how the function is used, and I can also paste those examples into the Console to see their output. Best way to learn what the function actually does is seeing it in action. Let's try:

```
sum(1:5)
```

So this is calculating the sum of the numbers from 1 and 5; that is what that `1:5` syntax means in this case. We can check it with the next example:

```
sum(1, 2, 3, 4, 5)
```

Awesome. Let's try this on our `mtcars` data

```
sum(mtcars)
```

Alright. What is this number? It is the sum of ALL of the data in the `mtcars` dataset. Maybe in some analysis this would be a useful operation, but I would worry about the way your data is set up and your analyses if this is ever something you'd want to do. More likely, you'd want to take the sum of a specific column. In R, you can do that with the `$` operator.

Let's say we want to calculate the total number of gears that all these cars have:

```
sum(mtcars$gear)
```

2.6 Deep thought

How would you do this in Excel? The calculations are usually the same shape as the data. In other words if you want to multiply 20 numbers stored in cells A1:An by 2, you will need 20 calculations: $=A1 * 2, =A2 * 2, \dots, =An * 2$.

OK so now that we've got a little bit of a feel for R and RStudio, let's do something much more interesting and really start feeling its power.

2.7 R Scripts

OK so working in the Console is great for quick things, but it gets messy. Keeping track at what I've done and trying to build upon it would be a nightmare.

Instead of working in the Console, we can be more organized by writing analyses in a script. This is a really powerful way to work in R. *TODO dev more* Scripts are a written record of the analyses you do, unlike Excel. And they can be re-run easily...

In this script, we're going to make our first figure in R. Let's all do this together.

File > New File > R Script.

This is a blank slate for us to write our code; but there are some good practices we can start off with. Let's add a useful header to the top of it. For example, at a minimum:

```
# -----  
# A descriptive title  
# Your name  
# Contact information  
# Date  
# -----
```

And then let's save it, naming it something like "my_first_figure.R". Let's get into good habits now with this filename: no spaces! Use underscores _ or dashes - or no space at all.

Since we're working in or Project, this script is now nicely saved in our Project. You can see our .R show up in our Files pane on the bottom right.

Let's attach a package. Since you've already installed tidyverse,

```
# Attach the tidyverse  
library(tidyverse)
```

What is the tidyverse? *TODO* - ggplot2

Let's look at one of the datasets that is built into the ggplot2 package. Type this into your R script:

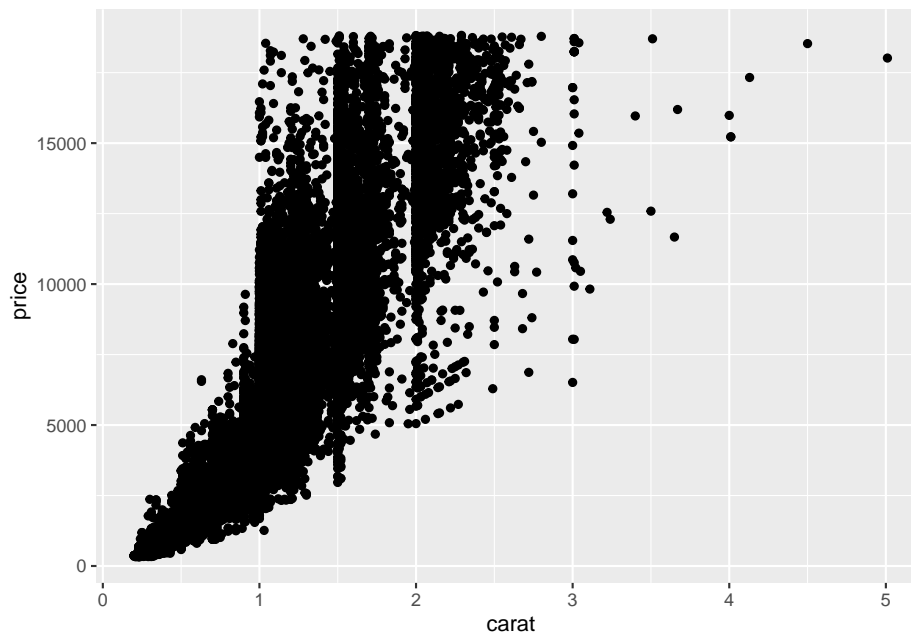
```
View(diamonds)
```

So this is not immediately executed like when we were typing in the Console. That's because an R script is really just a text file that doesn't do anything on its own; you need to tell R to execute it. You do that in a few ways (let's do each of them):

1. copy-paste this line into the console.
2. click Run (with green arrow at the top-right of your script) to run the line where your cursor is or any highlighted selection
3. click Source (top right of your script) to run the whole script.

Now let's plot it. Type this or copy-paste and then we'll discuss:

```
ggplot(diamonds, aes(x = carat, y = price)) +  
  geom_point()
```



2.7.1 Deep thought: Error messages are your friends

As Jenny Bryan says:

Implicit contract with the computer / scripting language: Computer will do tedious computation for you. In return, you will be completely precise in your instructions. Typos matter. Case matters. Pay attention to how you type.

Remember that this is a language, not unsimilar to English! There are times you aren't understood – it's going to happen. There are different ways this can happen. Sometimes you'll get an error. This is like someone saying 'What?' or 'Pardon'? Error messages can also be more useful, like when they say 'I didn't understand what you said, I was expecting you to say blah'. That is a great type of error message. Error messages are your friend. Google them (copy-and-paste!) to figure out what they mean.

And also know that there are errors that can creep in more subtly, when you are giving information that is understood, but not in the way you meant. Like if I am telling a story about suspenders that my British friend hears but silently interprets in a very different way (true story). This can leave me thinking I've gotten something across that the listener (or R) might silently interpreted very differently. And as I continue telling my story you get more and more confused... Clear communication is critical when you code: write clean, well documented code and check your work as you go to minimize these circumstances!

2.8 Don't save the workspace

2.9 Deep thought: keep the raw data raw.

Discussing using Excel for variables.

Horror Stories! Economist etc. –

Mine: genetics example doesn't hit home as much as Durham bike accidents where age groups are converted to dates just by opening the bloody csv in excel

2.10 Activity 1

2.11 Activity 2

2.12 Efficiency Tips

2.13 Troubleshooting

Here are some additional things we didn't have time to discuss:

2.13.1 I entered a command and nothing's happening

It may be because you didn't complete a command: is there a little + in your console? R is saying that it is waiting for you to finish. In the example below, I need to close that parenthesis.

```
> x <- seq(1, 10  
+
```


Chapter 3

readxl

3.1 Summary

Check this, may need to be a block quote: The **readxl** package makes it easy to import tabular data from Excel spreadsheets (.xls or .xlsx files) and includes several options for cleaning data during import. **readxl** has no external dependencies and functions on any operating system, making it an OS- and user-friendly package that simplifies getting your data from Excel into R.

3.2 Objectives

- Use `readr::read_csv()` to read in a comma separated value (CSV) file
- Use `readxl::read_excel()` to read in an Excel worksheet from a workbook
- Replace a specific string/value in a spreadsheet with `NA`
- Skip *n* rows when importing an Excel worksheet
- Use `readxl::read_excel()` to read in parts of a worksheet (by cell range)
- Specify column names when importing Excel data
- Read and combine data from multiple Excel worksheets into a single df using `purrr::map_df()`
- Write data using `readr::write_csv()` or `writexl::write_excel()`
- Workflows with **readxl**: considerations, limitations, reproducibility

3.3 Resources

- <https://readxl.tidyverse.org/>

- readxl Workflows article (from tidyverse.org)

3.4 Lesson

3.4.1 Lesson prep: get data files into your working directory

In Session 1, we introduced how and why R Projects are great for reproducibility, because our self-contained working directory will be the **first** place R looks for files.

You downloaded four files for this workshop:

- fish_counts_curated.csv
- invert_counts_curated.xlsx
- kelp_counts_curated.xlsx
- substrate_cover_curated.xlsx

Copy and paste those files into the ‘r-and-excel’ folder on your computer. Notice that now these files are in your working directory when you go back to that Project in RStudio (check the ‘Files’ tab). That means they’re going to be in the first place R will look when you ask it to find a file to read in.

3.4.2 Create a new .R script, attach the tidyverse, readxl and writexl packages

In your RforExcelUsers project in RStudio, open a new .R script and add a useful header to the top of it. For example, at a minimum:

```
# -----
# A descriptive title
# Summary of what this script is for
# Your name
# Contact information
# -----

# Other things you might include: required packages or datasets, relevant links (e.g.
```

In this lesson, we’ll read in a CSV file with the `readr::read_csv()` function, so we need to have the `readr` package attached. Since it’s part of the `tidyverse`, we’ll go ahead and attach the `tidyverse` package below our script header using `library(package_name)`. It’s a good idea to attach all necessary packages near the top of a script, so we’ll also attach the `readxl` packages here.

```
# Attach the tidyverse, readxl and writexl packages:
library(tidyverse)
library(readxl)
library(writexl)
```

Now, all of the packages and functions within the `tidyverse` and `readxl`, including `readr::read_csv()` and `readxl::read_excel()`, are available for use.

3.4.3 Use `readr::read_csv()` to read in data from a CSV file

There are many types of files containing data that you might want to work with in R. A common one is a comma separated value (CSV) file, which contains values with each column entry separated by a comma delimiter. CSVs can be opened, viewed, and worked with in Excel just like an `.xls` or `.xlsx` file - but let's learn how to get data directly from a CSV into R where we can work with it more reproducibly.

The CSV we'll read in here is called "`fish_counts_curated.csv`", and contains observations for "the abundance and size of fish species as part of SBCLTER's kelp forest monitoring program to track long-term patterns in species abundance and diversity" from the Santa Barbara Channel Long Term Ecological Research program.

Source: Reed D. 2018. SBC LTER: Reef: Kelp Forest Community Dynamics: Fish abundance. Environmental Data Initiative. <https://doi.org/10.6073/pasta/dbd1d5f0b225d903371ce89b09ee7379>. Dataset accessed 9/26/2019.

Read in the "`fish_counts_curated.csv`" file `read_csv("file_name.csv")`, and store it in R as an object called `fish_counts`:

```
fish_counts <- read_csv("fish_counts_curated.csv")
```

Notice that the name of the stored object (here, `fish_counts`) will show up in our Environment tab in RStudio.

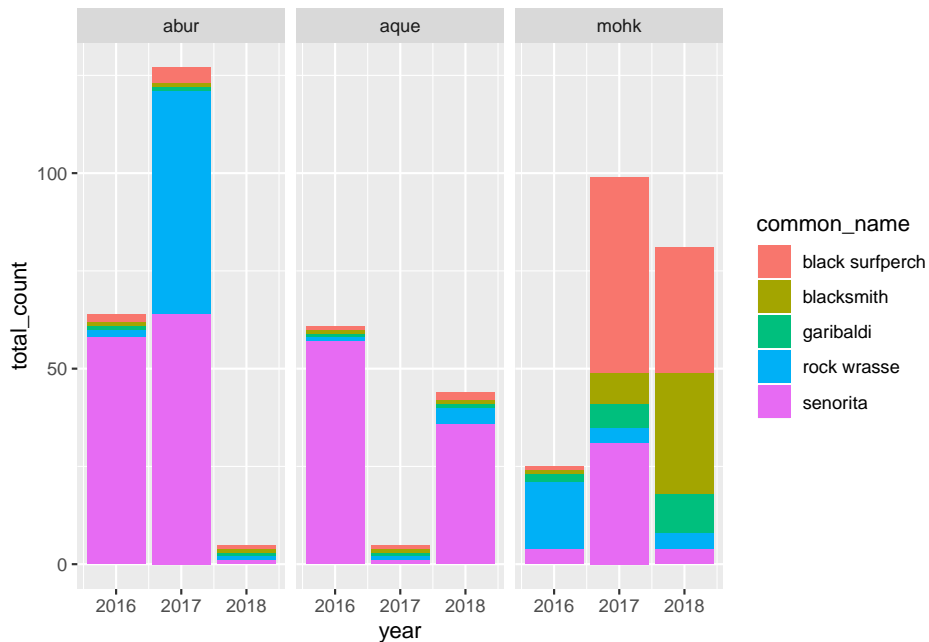
Click on the object in the Environment, and R will automatically run the `View()` function for you to pull up your data in a separate viewing tab. Now we can look at it in the spreadsheet format we're used to.

Here are a few other functions for quickly exploring imported data:

- `summary()`: summary of class, dimensions, NA values, etc.
- `names()`: variable names (column headers)
- `ls()`: list all objects in environment
- `head()`: Show the first x rows (default is 6 lines)
- `tail()`: Show the last x rows (default is 6 lines)

Now let's make a simple plot of some fish counts with `ggplot2`.

```
ggplot(fish_counts, aes(x = year, y = total_count)) +
  geom_col(aes(fill = common_name)) +
  facet_wrap(~site)
```



Now that we have our fish counts data ready to work with in R, let's get the substrate cover and kelp data (both .xlsx files). In the following sections, we'll learn that we can use `readxl::read_excel()` to read in Excel files directly.

3.4.4 Use `readxl::read_excel()` to read in a single Excel worksheet

First, take a look at `substrate_cover_curated.xlsx` in Excel, which contains a single worksheet with substrate type and percent cover observations at different sampling locations in the Santa Barbara Channel.

A few things to notice:

- The file contains a single worksheet
- There are multiple rows containing text information up top
- Where observations were not recorded, there exists '-9999'

Let's go ahead and read in the data. If the file is in our working directory, we can read in a single worksheet .xlsx file using `readxl::read_excel("file_name.xlsx")`. *Note: `readxl::read_excel()` works for both .xlsx and .xls types.*

Like this:


```
substrate_cover <- read_excel("substrate_cover_curated.xlsx")
```

Tada? Not quite.

Click on the object name (*substrate_cover*) in the Environment to view the data in a new tab. A few things aren't ideal:

```
substrate_cover

## # A tibble: 23,942 x 9
##   `Substrate cover data` ...2 ...3 ...4 ...5 ...6 ...7 ...8 ...9
##   <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
## 1 Source: https://porta~ <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
## 2 Accessed: 9/28/2019 <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
## 3 <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
## 4 year month date site trans~ quad side subs~ perc~
## 5 -9999 -9999 -9999 carp 1 20 i b -9999
## 6 2000 9 -9999 carp 1 20 o b -9999
## 7 2000 9 9/8/00 carp 1 20 i b 100
## 8 2000 9 9/8/00 carp 1 20 o b 100
## 9 2000 9 9/8/00 carp 1 40 i b 100
## 10 2000 9 9/8/00 carp 1 40 o b 100
## # ... with 23,932 more rows
```

- The top row of text has automatically become the (messy) column headers
- There are multiple descriptive rows before we actually get to the data
- There are -9999s that we want R to understand NA instead

We can deal with those issues by adding arguments within `read_excel()`. Include argument `skip = n` to skip the first 'n' rows when importing data, and `na = "this"` to replace "this" with NA when importing:

```
substrate_cover <- read_excel("curation/substrate_cover_curated.xlsx", skip = 4, na = "-9999")
```

```
substrate_cover

## # A tibble: 23,938 x 9
##   year month date site transect quad side substrate_type
##   <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
## 1 <NA> <NA> <NA> carp 1 20 i b
## 2 2000 9 <NA> carp 1 20 o b
## 3 2000 9 9/8/~ carp 1 20 i b
## 4 2000 9 9/8/~ carp 1 20 o b
## 5 2000 9 9/8/~ carp 1 40 i b
## 6 2000 9 9/8/~ carp 1 40 o b
## 7 2000 9 9/8/~ carp 2 20 i b
## 8 2000 9 9/8/~ carp 2 20 o b
## 9 2000 9 9/8/~ carp 2 40 i b
## 10 2000 9 9/8/~ carp 2 40 o b
```

```
## # ... with 23,928 more rows, and 1 more variable: percent_cover <chr>
```

Check out *substrate_cover*, and see that the first row *after* the 4 skipped are the column names, and all -9999s have been updated to NA. Hooray!

3.4.5 Use `readxl::read_excel()` to read in only *part* of an Excel worksheet

We always advocate for leaving the raw data raw, and writing a complete script containing all steps of data wrangling & transformation. But in *some* situations (be careful), you may want to specify a range of cells to read in from an Excel worksheet.

You can specify a range of cells to read in using the `range =` argument in `read_excel()`. For example, if I want to read in the rectangle from D12:I15 in *substrate_cover_curated.xlsx* - only observations for Carpenteria Beach (Transect 2) in September 2000 - I can use:

```
carp_cover_2000 <- readxl::read_excel("substrate_cover_curated.xlsx", range = "D12:I15")
```

But yuck. Look at *carp_cover_2000* and you'll notice that the first row *of that range* is automatically made the column headers. To keep all rows within a range and **add your own column names**, add a `col_names =` argument:

```
carp_cover_2000 <- readxl::read_excel("substrate_cover_curated.xlsx", range = "D12:I15", col_names =
```

```
carp_cover_2000
```

```
## # A tibble: 4 x 6
##   site_name transect quad plot_side type coverage
##   <chr>      <chr>   <chr> <chr>   <chr> <chr>
## 1 carp      2      20    i      b     90
## 2 carp      2      20    o      b     80
## 3 carp      2      40    i      b     80
## 4 carp      2      40    o      b     85
```

So far we've read in a single CSV file using `readr::read_csv()`, and an Excel file containing a single worksheet with `readxl::read_excel()`. Now let's read in data from an Excel workbook with multiple worksheets.

3.4.6 Use `readxl::read_excel()` to read in selected worksheets from a workbook

Now, we'll read in the kelp fronds data from file *kelp_counts_curated.xlsx*. If you open the Excel workbook, you'll see that it contains multiple worksheets with giant kelp observations in the Santa Barbara Channel during July 2016, 2017, and 2018, with data collected at each *site* in a separate worksheet.

To read in a single Excel worksheet from a workbook we'll again use `readxl::read_excel("file_name.xlsx")`, but we'll need to let R know which worksheet to get.

Let's read in the kelp data just like we did above, as an object called *kelp_counts*.

```
kelp_counts <- readxl::read_excel("kelp_counts_curated.xlsx")
```

You might be thinking, "Hooray, I got all of my Excel workbook data!" But remember to always look at your data - you will see that actually only the first worksheet was read in. The default in `readxl::read_excel()` is to read in the **first worksheet** in a multi-sheet Excel workbook.

To check the worksheet names in an Excel workbook, use `readxl::excel_sheets()`:

```
readxl::excel_sheets("kelp_counts_curated.xlsx")
```

If we want to read in data from a worksheet other than the first one in an Excel workbook, we can specify the correct worksheet by name or position by adding the `sheet` argument.

Let's read in data from the worksheet named *golb* (Goleta Beach) in the *kelp_counts_curated.xlsx* workbook:

```
kelp_golb <- readxl::read_excel("kelp_counts_curated.xlsx", sheet = "golb")
```

Note that you can also specify a worksheet by position: since *golb* is the 6th worksheet in the workbook, we could also use the following:

```
kelp_golb <- readxl::read_excel("kelp_counts_curated.xlsx", sheet = 6)
```

```
kelp_golb
```

```
## # A tibble: 3 x 4
##   year month site  total_fronds
##   <chr> <chr> <chr>         <dbl>
## 1 2016   7    golb           2557
## 2 2017   7    golb           1575
## 3 2018   7    golb           1629
```

3.4.7 Read in and combine data from multiple worksheets into a data frame simultaneously with `purrr::map_df()`

So far, we've read in entire Excel worksheets and pieces of a worksheet. What if we have a workbook (like *kelp_counts_curated.xlsx*) that contains worksheets that contain observations for the same variables, in the same organization? Then we may want to read in data from *all* worksheets, and combine them into a single data frame.

We'll use `purrr::map_df()` to loop through all the worksheets in a workbook, reading them in & putting them together into a single df in the process.

The steps we'll go through in the code below are:

- Set a pathway so that R knows where to look for an Excel workbook
- Get the names of all worksheets in that workbook with `excel_sheets()`
- Set names of a vector with `set_names()`
- Read in all worksheets, and put them together into a single data frame with `purrr::map_df()`

QUESTION: Have they learned the pipe operator at this point?

Expect the question: Why do I need to use `read_excel()` instead of just giving it the file path (as below)?

```
kelp_path <- "kelp_counts_curated.xlsx"

kelp_all_sites <- kelp_path %>%
  excel_sheets() %>%
  set_names() %>%
  purrr::map_df(read_excel, kelp_path)
```

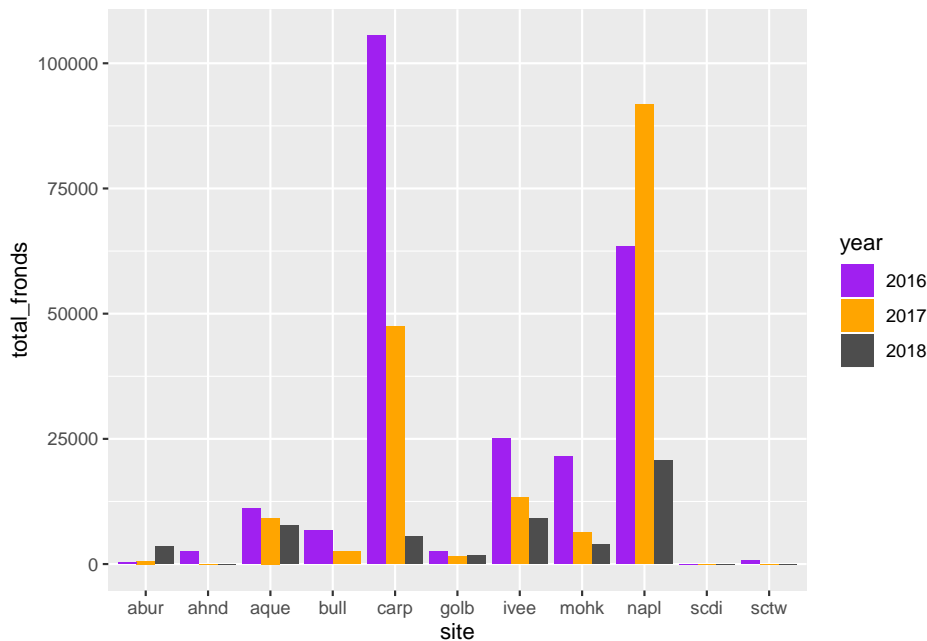
Check out `kelp_all_sites`, and notice that now the data from all 11 sites is now collected into a single data frame:

```
kelp_all_sites

## # A tibble: 32 x 4
##   year month site total_fronds
##   <chr> <chr> <chr>         <dbl>
## 1 2016   7   abur             307
## 2 2017   7   abur             604
## 3 2018   7   abur            3532
## 4 2016   7   ahnd             2572
## 5 2017   7   ahnd              16
## 6 2018   7   ahnd              16
## 7 2016   7   aque            11152
## 8 2017   7   aque             9194
## 9 2018   7   aque             7754
## 10 2016   7   bull             6706
## # ... with 22 more rows
```

Awesome! Let's make a graph with `ggplot2`:

```
ggplot(kelp_all_sites, aes(x = site, y = total_fronds)) +
  geom_col(aes(fill = year), position = "dodge") +
  scale_fill_manual(values = c("purple", "orange", "gray30"))
```



3.4.8 Save data frames as .csv or .xlsx with `readr::write_csv()` or `writexl::write_xlsx()`

There are a number of reasons you might want to save (/export) data in a data frame as a .csv or Excel worksheet, including:

- To cache raw data within your project
- To store copies of intermediate data frames
- To convert your data back to a format that your coworkers/clients/colleagues will be able to use it more easily

Use `readr::write_csv(object, "file_name.csv")` to write a data frame to a CSV, or `writexl::write_xlsx(object, "file_name.xlsx")` to similarly export as a .xlsx (or .xls) worksheet.

In the previous step, we combined all of our kelp frond observations into a single data frame. Wouldn't it make sense to store a copy?

As a CSV:

```
readr::write_csv(kelp_all_sites, "kelp_all_sites.csv")
```

A cool thing about `readr::read_csv()` is that it just quietly *works* without wrecking anything else you do in a sequence, so it's great to add at the end of a piped sequence.

For example, if I want to read in the ‘ivee’ worksheet from `kelp_counts_curated.xlsx`, select only columns ‘year’ and ‘total_fronds’, then write that new subset to a .csv file, I can pipe all the way through:

```
kelp_ivee <- readxl::read_excel("kelp_counts_curated.xlsx", sheet = "ivee") %>%
  select(year, total_fronds) %>%
  write_csv("kelp_ivee.csv")
```

Now I’ve created `kelp_ivee.csv`, but the object `kelp_ivee` also exists for me to use in R.

If needed, I can also export a data frame as an Excel (.xlsx) worksheet:

```
writexl::write_xlsx(kelp_all_sites, "kelp_all_sites.xlsx")
```

3.5 Fun facts ideas:

- Did you know that Clippy shows up to help you in the documentation for `?writexl::write_xlsx()`?
- The name of the `purrr` package? Why map?

3.6 Interludes (deep thoughts/openscapes)

- Workflow/reproducibility/readxl workflows article
- Respecting the tools people are working with already (e.g. don’t make your Excel using co-workers hate you)

3.7 Activity: Import some invertebrates!

There’s one dataset we haven’t imported or explored yet: invertebrate counts for 5 popular invertebrates (California cone snail, California spiny lobster, orange cup coral, purple urchin and rock scallops) at 11 sites in the Santa Barbara Channel. Take a look at the `invert_counts_curated.xlsx` data by opening it in Excel

- Read in the `invert_counts_curated.xlsx` worksheet as object ‘`inverts_july`’, only retaining **site**, **common_name**, and **2016** and setting the existing first row in the worksheet as to column headers upon import
- Explore the imported data frame using `View`, `names`, `head`, `tail`, etc.
- Write ‘`inverts_july`’ to a CSV file in your working directory called “`inverts_july.csv`”
- Create a basic graph of invert counts in 2016 (y-axis) by site (x-axis), with each species indicated by a different fill color

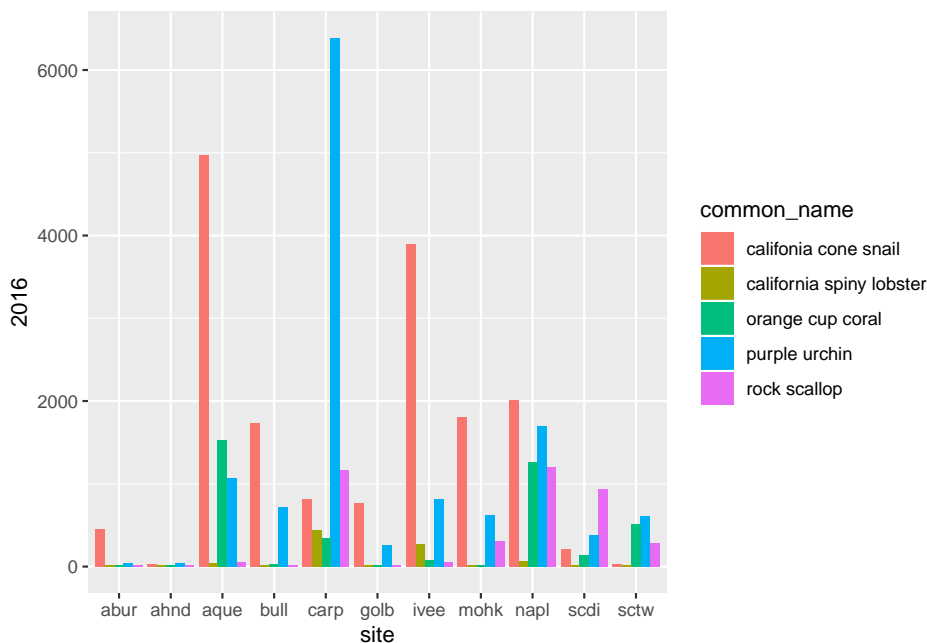
- **Note:** If your column name is a number (not great) you'll probably want to rename it...but in the meantime, to call it as a variable make sure you put single or double quotes around it (e.g. '2016' or "2016") so that R recognizes it's a variable and not a value

```
# Importing only 'site' through '2016' columns:
inverts_july <- readxl::read_excel("curation/invert_counts_curated.xlsx", range = "B1:D56")

# Do some basic exploring (why might we want to do this in the Console instead?):
#View(inverts_july)
names(inverts_july)
head(inverts_july)
tail(inverts_july)
ls()

# Make a gg-graph plot:
inverts_graph <- ggplot(inverts_july, aes(x = site, y = `2016`)) +
  geom_col(aes(fill = common_name),
           position = "dodge")
```

```
inverts_graph
```



3.8 Efficiency Tips

- Add an assignment arrow in script/code chunk (<-): Alt + minus (-)
- Undo shortcut: Command + Z
- Redo shortcut: Command + Shift + Z

Chapter 4

RMarkdown

4.1 Summary (a few sentences)

Here we'll learn RMarkdown, which helps you tell a story with your data analysis because you can write text alongside the code.

4.2 Objectives (more detailed, bulletpoints?)

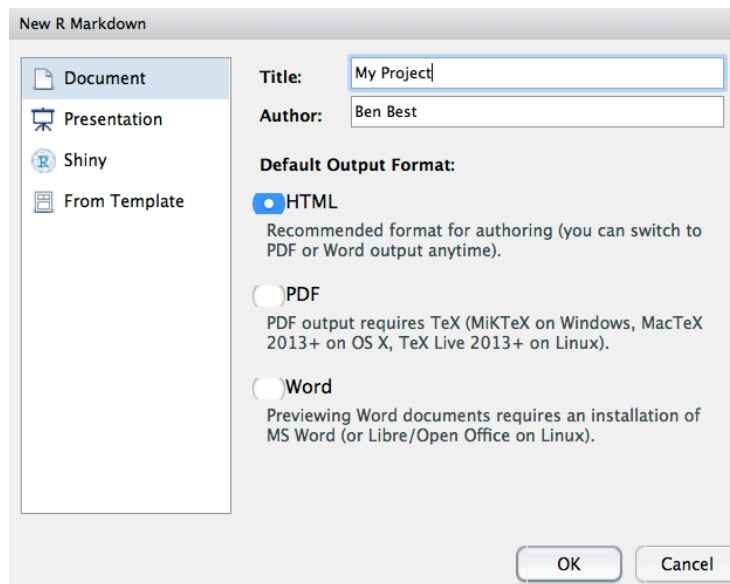
4.3 Resources

4.4 Intro to RMarkdown

TODO (write out): Intro, knitting How it's different from R Script (which we just created) How in-line figures are awesome (no copy-pasting from excel to word) Reproducibility is important (this might be new to some people). Reproducible research/self-contained Rmd files: read in data at the top. Read_csv.

An RMarkdown file will allow us to weave markdown text with chunks of R code to be evaluated and output content like tables and plots.

File -> New File -> RMarkdown... -> Document of output format HTML, OK.



You can give it a Title like “My Analysis”. Then click OK.

OK, first off: by opening a file, we are seeing the 4th pane of the RStudio console, which is essentially a text editor. This lets us organize our files within RStudio instead of having a bunch of different windows open.

Let’s have a look at this file — it’s not blank; there is some initial text is already provided for you. Notice a few things about it:

- There are white and grey sections. R code is in grey sections, and other text is in white.

```

1 ---
2 title: "My Project"
3 author: "Julie"
4 date: "11/21/2017"
5 output: html_document
6 ---
7
8 ```{r setup, include=FALSE}
9 knitr::opts_chunk$set(echo = TRUE)
10 ```
11
12 ## R Markdown
13
14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word
15 documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.
16
17 When you click the Knit button a document will be generated that includes both content as well as the
18 output of any embedded R code chunks within the document. You can embed an R code chunk like this:
19
20 ```{r cars}
21 summary(cars)
22 ```
23
24 ## Including Plots
25
26 You can also embed plots, for example:
27
28 ```{r pressure, echo=FALSE}
29 plot(pressure)
30 ```
31
32 Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that
33 generated the plot.

```

Let's go ahead and “Knit HTML” by clicking the blue yarn at the top of the RMarkdown file.



What do you notice between the two? So much of learning to code is looking for patterns.

Notice how the grey **R code chunks** are surrounded by 3 backticks and `{r LABEL}`. These are evaluated and return the output text in the case of `summary(cars)` and the output plot in the case of `plot(pressure)`.

Notice how the code `plot(pressure)` is not shown in the HTML output because of the R code chunk option `echo=FALSE`.

The hashtag (#) at lines 12 and 22 cause the following text to be displayed larger and in bold.

This RMarkdown file has 2 different languages within it: **R** and **Markdown**.

We don't know that much R yet, but you can see that we are taking a summary of some data called 'cars', and then plotting. We will focus on R for the rest of the workshop, but for the rest of this morning let's focus on the second language.

The second language is Markdown. This is a formatting language for plain text, and there are only about 15 rules to know.

Notice the syntax for:

- **headers** get rendered at multiple levels: #, ##
- **bold**: **word**

There are some good cheatsheets to get you started, and here is one built into RStudio: Go to Help > Markdown Quick Reference

Important: note that the hashtag # is used differently in Markdown and in R:

- in R, a hashtag indicates a comment that will not be evaluated. You can use as many as you want: # is equivalent to #####. It's a matter of style. I use two ## to indicate a comment so that it's clearer what is a comment versus what I don't want to run at the moment.
- in Markdown, a hashtag indicates a level of a header. And the number you use matters: # is a “level one header”, meaning the biggest font and

the top of the hierarchy. `###` is a level three header, and will show up nested below the `#` and `##` headers.

If this seems confusing, take comfort in the fact that you are already used to using `#`s differently in real life: it can mean “number” or “pound” or hashtags on social media.

Learn more: <http://rmarkdown.rstudio.com/>

4.4.1 Activity

1. In Markdown write some italic text, make a numbered list, and add a few subheaders. Use the Markdown Quick Reference (in the menu bar: Help > Markdown Quick Reference).
2. Reknit your html file.

4.4.2 Code chunks

OK. Now let's practice with some of those commands that we were working on this morning.

Create a new chunk in your RMarkdown first in one of these ways:

- click “Insert > R” at the top of the editor pane
- type by hand “`{r}`”
- if you haven't deleted a chunk that came with the new file, edit that one

Now, let's write some R code.

```
x <- seq(1:15)
```

Now, hitting return does not execute this command; remember, it's a text file in the text editor, it's not associated with the R engine. To execute it, we need to get what we typed in the the R chunk (the grey R code) down into the console. How do we do it? There are several ways (let's do each of them):

1. copy-paste this line into the console.
2. select the line (or simply put the cursor there), and click ‘Run’. This is available from
 - a. the bar above the file (green arrow)
 - b. the menu bar: Code > Run Selected Line(s)
 - c. keyboard shortcut: command-return
3. click the green arrow at the right of the code chunk

4.4.3 Activity

Add a few more commands to your file from this morning. Execute them by trying the three ways above. Then, knit your R Markdown file, which will also save the Rmd by default.

4.5 Create a Rmd for our fish analysis

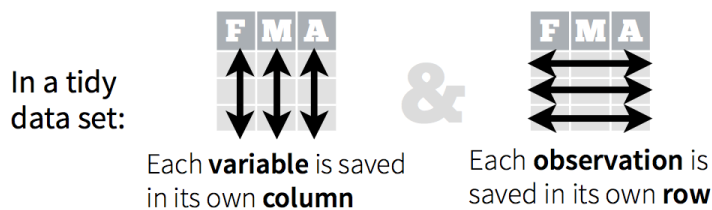
OK now let's open a new RMarkdown file and begin some analyses.

- read in
- View()

fish_counts are what we call Tidy Data.

4.6 Tidy data

Tidy data has a simple convention: put variables in the columns and observations in the rows.



TODO: Explore data: summaries/data table Uncount and count Why would you ever want to uncount? You don't want to assume how future you will want to use this data How easy is it to get these counts by different groupings Simplest ggplot (introduced in readxl chapter)

4.7 Activity

1. knit!

4.8 Fun facts (quirky things) - making a note of these wherever possible for interest (little “Did you know?” sections)

4.9 Interludes (deep thoughts/openscapes)

4.10 Interludes (deep thoughts/openscapes)

Comments! Organization (spacing, subsections, vertical structure, indentation, etc.)! Well-named variables! Also, well-named operations so analyses (`select(data, columnname)`) instead of `data[1:6,5]` and excel equivalent. (Ex with strings) Not so brittle/sensitive to minor changes.

4.10.1 RMarkdown video (1-minute)

Let’s watch this to demonstrate all the amazing things you can now do:

What is RMarkdown?

4.11 Efficiency Tips

Chapter 5

dplyr and Pivot Tables

5.1 Summary (a few sentences)

First of two chapters on data wrangling, here focused on pivot tables.

5.2 Objectives (more detailed, bulletpoints?)

In R, we can use dplyr for pivot tables by using 2 main verbs in combination: `group_by` and `summarize`, and that's where we'll start. Then, we will learn 2 critical verbs that are powerful for data wrangling: `mutate` and `select`.

We will also continue to emphasize reproducibility in all our analyses.

5.3 Resources

- dplyr.tidyverse.org
- R for Data Science: Transform Chapter

5.4 Introduction / our analytical plan

We are going to continue with our analysis with the fish-counts data. So far, our RMarkdown has the following in the “setup” chunk:

```
## attach libraries  
library(tidyverse)
```

```
## read in data
fish_counts <- read_csv("fish_counts_curated.csv")
```

And we have explored the data by looking at some summary statistics and making a simple plot. Now, we are going to learn how to wrangle data in R, using the `dplyr` package which is included in the `tidyverse`. In this session, we'll focus on the functions in `dplyr` that operate like pivot tables.

This is because what we want to do with our `fish_counts` data is XXXXX.

5.5 What are pivot tables?

TODO:

- what they are
- what they allow you to do

Let's talk about how this looks like in R.

5.6 dplyr overview

`dplyr` is a grammar of data manipulation that provides a consistent set of verbs that help you solve the most common data manipulation challenges. These common verbs are:

- **filter()**: pick observations by their values
- **select()**: pick variables by their names
- **mutate()**: create new variables with functions of existing variables
- **summarise()**: collapse many values down to a single summary
- **arrange()**: reorder the rows

These can all be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the verbs for a language of data manipulation.

All verbs work similarly:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame. You can refer to columns in the data frame directly without using `$`.
3. The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result using the pipe operator `%>%`.

I love thinking of these `dplyr` verbs and the pipe operator `%>%` as telling a story. When I see `%>%` I think “and then”:

```
data %>%           # start with data, and then
  group_by() %>%   # group by a variable, and then
  mutate() %>%    # mutate to add a new column, and then
  select()         # select specific columns
```

5.7 pivot tables: `group_by()` `%>%` `summarize()`

In R, we can create the functionality of pivot tables by using 2 main `dplyr` verbs in combination: `group_by` and `summarize`.

Say it with me: “pivot tables are `group_by` and then `summarize`”. And just like pivot tables, you have flexibility with how you are going to summarize. For example, we can calculate an average, or a total.

Let’s try this on our `fish_counts` data. Let’s summarize this data by calculating the the total number of fish by type, which in our dataframe is called `common_name`. We’ll use the pipe operator `%>%`

```
fish_counts %>%
  group_by(common_name) %>%
  summarize(total_fish = sum(total_count)) # within summarize, we name a new column and calculate
```

```
## # A tibble: 5 x 2
##   common_name    total_fish
##   <chr>         <dbl>
## 1 black surfperch      94
## 2 blacksmith          46
## 3 garibaldi           24
## 4 rock wrasse         91
## 5 senorita           256
```

This returns output summarizing the `total_fish` for each fish. There are way more `senorita` fish than any other kind!

Notice how together, `group_by` and `summarize` collapse the amount of information we see. We lose the other columns that aren’t involved here; we no longer have `year`, `common_name`, or `total_count`. This is also what you would expect from a pivot table.

Question: What if you *don’t* `group_by` first? Let’s try it and discuss what’s going on.

```
fish_counts %>%
  summarize(total_fish = sum(total_count))
```

```
## # A tibble: 1 x 1
##   total_fish
##       <dbl>
## 1         511
```

So if we don't `group_by` first, we will get a single summary statistic (total in this case) for the whole dataset. Useful in some cases for sure. But being able to do it so easily by group can be very powerful.

Let's now check the `fish_counts` variable.

```
View(fish_counts)
```

We see that we haven't changed any of our original data that was stored in this variable. But maybe we do want to save this summary as a variable so we can refer to it or even include it in further analyses. So let's add a variable assignment to that first line:

```
common_name_summary <- fish_counts %>%
  group_by(common_name) %>%
  summarize(total_fish = sum(total_count))
```

Now let's summarize by site. We will assign it to a variable called `site_summary`. How do we do that? Let's do it together:

```
site_summary <- fish_counts %>%
  group_by(site) %>%
  summarize(total_fish = sum(total_count))
```

Great. It can be useful to summarize by both site and year, so that we can learn a little more about how things change over time across sites. And, awesomely, we are able to `group_by` more than one variable. Let's do this together, and assign this to a new variable called `site_year_summary`:

```
site_year_summary <- fish_counts %>%
  group_by(site, year) %>%
  summarize(total_fish = sum(total_count))
```

```
site_year_summary
```

```
## # A tibble: 9 x 3
## # Groups:   site [3]
##   site  year total_fish
##   <chr> <dbl>      <dbl>
## 1 abur  2016         64
## 2 abur  2017        127
```

```
## 3 abur 2018 5
## 4 aque 2016 61
## 5 aque 2017 5
## 6 aque 2018 44
## 7 mohk 2016 25
## 8 mohk 2017 99
## 9 mohk 2018 81
```

OK, so this is awesome. We can see the total counts for each site by year, and have this saved here in a nice variable.

We will revisit this in a moment, but now let's move on to our next `dplyr` verb.

5.8 `mutate()`

We use the `mutate()` function to add columns to a data frame. This is one of the most common things that I do in Excel: you need to name the new column, and then you can fill it with new values. From the help pages, we learn that unlike `summarize()`, `mutate()` preserves the number of rows of the input. Additionally, new variables overwrite existing variables of the same name.

Let's say we need to add a column that indicates that these observations were made by SCUBA diving. To do this, first we tell R we want to add a new column using the `mutate()` function. Then, we tell it the name of the column we want, let's call it `observation_type`. Then, we tell it the value we want in the cells: let's say "SCUBA". We need to put SCUBA in quotes:

```
fish_counts %>%
  mutate(observation_type = "SCUBA")
```

```
## # A tibble: 45 x 5
##   year site common_name total_count observation_type
##   <dbl> <chr> <chr>          <dbl> <chr>
## 1 2016 abur black surfperch      2 SCUBA
## 2 2016 abur blacksmith        1 SCUBA
## 3 2016 abur garibaldi         1 SCUBA
## 4 2016 abur rock wrasse       2 SCUBA
## 5 2016 abur senorita        58 SCUBA
## 6 2016 aque black surfperch    1 SCUBA
## 7 2016 aque blacksmith        1 SCUBA
## 8 2016 aque garibaldi         1 SCUBA
## 9 2016 aque rock wrasse       1 SCUBA
## 10 2016 aque senorita        57 SCUBA
## # ... with 35 more rows
```

Notice that when you just give one value like "SCUBA", `mutate` will repeat this value for you; it's the equivalent in Excel to when you grab the bottom right

corner of a cell and drag down.

Let's try a calculation. Let's calculate the total count for the whole data site as we did above. We add a new column named `total_fish`:

```
fish_counts %>%
  mutate(total_fish = sum(total_count))

## # A tibble: 45 x 5
##   year site common_name total_count total_fish
##   <dbl> <chr> <chr>         <dbl>     <dbl>
## 1  2016 abur black surfperch         2         511
## 2  2016 abur blacksmith          1         511
## 3  2016 abur garibaldi           1         511
## 4  2016 abur rock wrasse         2         511
## 5  2016 abur senorita          58         511
## 6  2016 aque black surfperch         1         511
## 7  2016 aque blacksmith          1         511
## 8  2016 aque garibaldi           1         511
## 9  2016 aque rock wrasse         1         511
## 10 2016 aque senorita          57         511
## # ... with 35 more rows
```

And notice that this was the same calculated value as when we did this with `summarize`, but here it is repeated for every row instead of being collapsed.

5.8.1 Activity

Take 3 minutes to add a new column to the data frame; discuss with your neighbor for ideas!

5.8.2 `group_by()` %>% `mutate()`

So there are many things you could add to a new column; but let's focus on how powerful `mutate` can be in combination with `group_by`. So just like we were just doing `group_by()` %>% `summarize()`, we can do `group_by()` %>% `mutate()`.

Let's add a new column named `siteyear_counts`, and we will calculate it after grouping by site and year. Let's have a look at it first, and then we will assign it as a variable in a moment.

```
fish_counts %>%
  group_by(site, year) %>%
  mutate(siteyear_counts = sum(total_count))

## # A tibble: 45 x 5
## # Groups:   site, year [9]
```

```
##      year site common_name      total_count siteyear_counts
##      <dbl> <chr> <chr>          <dbl>          <dbl>
##  1  2016 abur  black surfperch          2             64
##  2  2016 abur  blacksmith          1             64
##  3  2016 abur  garibaldi          1             64
##  4  2016 abur  rock wrasse         2             64
##  5  2016 abur  senorita          58             64
##  6  2016 aque  black surfperch          1             61
##  7  2016 aque  blacksmith          1             61
##  8  2016 aque  garibaldi          1             61
##  9  2016 aque  rock wrasse         1             61
## 10  2016 aque  senorita          57             61
## # ... with 35 more rows
```

We now have an additional column in our dataframe called `siteyear_counts`. And again, if we recall from our `site_year_summary` above, it has calculated the same information. But instead of collapsing our dataframe, we retain all of the information from the other columns, and the `siteyear_counts` column will have values that are repeated.

5.9 mutate() vs summarize()

Why would you use `mutate` instead of `summarize`? Why would you ever want to have that `siteyear_counts` column with values repeated like we just did? Why wouldn't you always do `group_by() %>% summarize()` rather than `group_by() %>% mutate()`? The truth is, there is no one way to do anything in R, but there are ways to make your analyses have fewer steps or read more nicely. Let's explore this by doing a bit of analysis.

Let's say we want to calculate the percentage of fish type at each site. This means we are going to do a calculation using both the raw and summary data. And we're going to do it in 2 ways, first using `group_by() %>% summarize()` and then `group_by() %>% mutate()`.

Let's start off doing this as a `group_by() %>% mutate()`.

```
fish_percs <- fish_counts %>%
  group_by(site, year) %>%
  mutate(siteyear_counts = sum(total_count)) %>%
  mutate(perc_fish = total_count/siteyear_counts*100)
```

When I'm doing analyses, I like `group_by() %>% mutate()` because I can build out the logic step-by-step and actually look at it as it builds. It's both comforting and good for error-checks; I can do what we call "spot checks" of calculating a few values by hand to make sure it's working. This would also be relatively easy for someone else to follow.

(Again here we can see that seniorita fish really dominate each site-year combination).

If we wanted to do this with `group_by()` `%>%` `summarize()` we would need a few more steps. We can write it up as pseudo-code:

```
## first calculate fish siteyear_counts
x <- fish_counts %>%
  group_by(site, year) %>%
  summarize(siteyear_counts = sum(total_count))

## then somehow join or merge that information to the fish_counts data
x %>%
  mutate(perc_fish = total_count/siteyear_counts*100)
```

In order to calculate the percentages with the appropriate values, we need to somehow join the summarized data back to the `fish_counts`. This actually requires a few more dplyr verbs: `filter` and `*_join`; we will do this tomorrow!

count column (would have to uncount first)

5.10 Deep thoughts

Highly recommended read: Broman & Woo: Data organization in spreadsheets. Practical tips to make spreadsheets less error-prone, easier for computers to process, easier to share

Great opening line: “Spreadsheets, for all of their mundane rectangularness, have been the subject of angst and controversy for decades.”

5.11 Efficiency Tips

arrow keys with shift, option, command

Chapter 6

Dplyr and vlookups

6.1 Summary

In Session 4, we learned how to do some basic wrangling and find summary information with functions in the `dplyr` package, which exists within the `tidyverse`. Those were:

TODO: Check this list (to see what actually gets covered in Session 4)

- `dplyr::select()`: select which **columns** to retain or exclude
- `dplyr::mutate()`: **add** a new column, while keeping the existing ones
- `dplyr::group_by()`: let R know that **groups** exist within the dataset, by variable(s)
- `dplyr::summarize()`: calculate a value (that you specify) for each group, then report each group's value in a table

In Session 5, we'll expand our data wrangling toolkit using:

- `dplyr::filter()` to conditionally subset our data by **rows**, and
- `dplyr::join()` functions to merge data frames together

The combination of `dplyr::filter()` and `dplyr::join()` - to return rows satisfying a condition we specify, and merging data frames by like variables - is analogous to the useful VLOOKUP function in Excel.

6.2 Objectives

- Continue building R Markdown skills
- Return **rows** that satisfy variable conditions using `dplyr::filter()`

- Use `dplyr::full_join()`, `dplyr::inner_join()`, and beyond to merge data frames by matching variables, with different endpoints in mind
- Use `dplyr::anti_join()` to find things that **do not** exist in both data frames
- Understand the similarities between `dplyr::filter()` + `join()` and Excel's VLOOKUP

6.3 Resources

- `dplyr::filter()` documentation from tidyverse.org
- `dplyr::join()` documentation from tidyverse.org
- Chapters 5 and 13 in *R for Data Science* by Garrett Golemund and Hadley Wickham

6.4 Lessons

Session 5 set-up: TODO

- Create a new .Rmd within the r-and-excel directory (project) you created in Session 1
- Add some descriptive text
- Add new code chunks to:
 - Attach packages
 - Read in the necessary data

In this session we'll use the `fish_counts_curated.csv` and `invert_counts_curated.xlsx` files, and the first worksheet from `kelp_counts_curated.xlsx`.

```
# Attach packages:
library(tidyverse)
library(readxl)

# Read in data:
invert_counts <- read_excel("invert_counts_curated.xlsx")
fish_counts <- read_csv("fish_counts_curated.csv")
kelp_counts_abur <- read_excel("kelp_counts_curated.xlsx")
```

Remember to always explore the data you've read in using functions like `View()`, `names()`, `summary()`, `head()` and `tail()` to ensure that the data you *think* you read in is *actually* the data you read in.

Now, let's use `dplyr::filter()` to decide which observations (rows) we'll keep or exclude in new subsets, similar to using Excel's VLOOKUP function.

6.4.1 `dplyr::filter()` to conditionally subset by rows

Use `dplyr::filter()` to let R know which **rows** you want to keep or exclude, based on what they contain. Some examples in words:

- “I only want to keep rows where the temperature is greater than 90°F.”
- “I want to keep all observations **except** those where the tree type is listed as **unknown**.”
- “I want to make a new subset with only data for mountain lions (the species variable) in California (the state variable).”

TODO: {Add filter image here}

When we use `dplyr::filter()`, we need to let R know a couple of things:

- What data frame we’re filtering from
- What condition(s) we want observations to **match** and/or **not match** in order to keep them in the new subset

Follow along with the examples below to learn some common ways to use `dplyr::filter()`.

6.4.1.1 Filter rows by matching a single character string

Let’s say we want to keep all observations from the `fish_counts` data frame where the common name is “garibaldi.” Here, we need to tell R to only *keep rows* from the **fish_counts** data frame when the common name (**common_name** variable) exactly matches **garibaldi**. Use `==` to ask R to look for matching strings:

```
fish_gari <- dplyr::filter(fish_counts, common_name == "garibaldi")
```

Check out the `fish_gari` object to ensure that only *garibaldi* observations remain.

You could also do this using the pipe operator `%>%` (though for a single function, it doesn’t save much effort or typing):

```
fish_gari <- fish_counts %>%
  dplyr::filter(common_name == "garibaldi")
```

6.4.1.2 Filter rows based on numeric conditions

Use expected operators (`>`, `<`, `>=`, `<=`, `=`) to set conditions for a numeric variable when filtering. For this example, we only want to retain observations when the **total_count** column value is `>= 50`:

```
fish_over50 <- dplyr::filter(fish_counts, total_count >= 50)
```

Or, using the pipe:

```
fish_over50 <- fish_counts %>%
  dplyr::filter(total_count >= 50)
```

TODO: show example of between and exact =

6.4.1.3 Filter to return rows that match *this* OR *that* OR *that*

What if we want to return a subset of the `fish_counts` df that contains *garibaldi*, *blacksmith* OR *black surfperch*?

There are several ways to write an “OR” statement for filtering, which will keep any observations that match Condition A *or* Condition B *or* Condition C. In this example, we will create a subset from `fish_counts` that only contains rows where the `common_name` is *garibaldi* or *blacksmith* or *black surfperch*.

Use `%in%` to ask R to look for *any matches* within a combined vector of strings:

```
fish_3sp <- fish_counts %>%
  dplyr::filter(common_name %in% c("garibaldi", "blacksmith", "black surfperch"))
```

Alternatively, you can indicate **OR** using the vertical line operator `|` to do the same thing (but you can see that it’s more repetitive when looking for matches within the same variable):

```
fish_3sp <- fish_counts %>%
  dplyr::filter(common_name == "garibaldi" | common_name == "blacksmith" | common_name
```

6.4.1.4 Filter to return rows that match conditions for multiple variables

In the previous examples, we set filter conditions based on a single variable (e.g. `common_name`). What if we want to return observations that satisfy conditions for multiple variables?

For example: We want to create a subset that only returns rows from ‘invert_counts’ where the `site` is “abur” or “mohk” *and* the `common_name` is “purple urchin.” In `dplyr::filter()`, add a comma (or ampersand, `&`) between arguments for multiple *AND* conditions:

```
urchin_abur_mohk <- invert_counts %>%
  dplyr::filter(site %in% c("abur", "mohk"), common_name == "purple urchin")

head(urchin_abur_mohk)
```

```
## # A tibble: 2 x 6
##   month site common_name `2016` `2017` `2018`
```

```
##   <chr> <chr> <chr>           <dbl> <dbl> <dbl>
## 1 7      abur  purple urchin      48    48    48
## 2 7      mohk  purple urchin     620   505   323
```

Like most things in R, there are other ways to do the same thing. For example, you could do the same thing using `&` (instead of a comma) between “and” conditions:

```
# Use the ampersand (&) to add another condition "and this must be true":

urchin_abur_mohk <- invert_counts %>%
  dplyr::filter(site %in% c("abur", "mohk") & common_name == "purple urchin")
```

Or you could just do two filter steps in sequence:

```
# Written as multiple filter steps:

urchin_abur_mohk <- invert_counts %>%
  dplyr::filter(site %in% c("abur", "mohk")) %>%
  dplyr::filter(common_name == "purple urchin")
```

6.4.1.5 Filter to return rows that *do not* match conditions

Sometimes we might want to exclude observations. Here, let’s say we want to make a subset that contains all rows from `fish_counts` except those recorded at the Mohawk Reef site (“mohk” in the `site` variable).

We use `!=` to return observations that **do not match** a condition.

Like this:

```
fish_no_mohk <- fish_counts %>%
  dplyr::filter(site != "mohk")
```

This similarly works to exclude observations by a value.

For example, if we want to return all observations *except* those where the total fish count is 1, we use:

```
fish_more_one <- fish_counts %>%
  dplyr::filter(total_count != 1)
```

What if we want to exclude observations for multiple conditions? For example, here we want to return all rows where the fish species **is not** garibaldi **or** rock wrasse.

We can use `filter(!variable %in% c("apple", "orange"))` to return rows where the variable does **not** match “apple” or “orange”. For our fish example, that looks like this:

```
fish_subset <- fish_counts %>%
  dplyr::filter(!common_name %in% c("garibaldi", "rock wrasse"))
```

Which then only returns observations for the other fish species in the dataset.

```
head(fish_subset)
```

```
## # A tibble: 6 x 4
##   year site common_name total_count
##   <dbl> <chr> <chr>          <dbl>
## 1  2016 abur black surfperch          2
## 2  2016 abur blacksmith            1
## 3  2016 abur senorita            58
## 4  2016 aque black surfperch          1
## 5  2016 aque blacksmith            1
## 6  2016 aque senorita            57
```

6.4.1.6 Example: Some combo w/previous functions used

We can also use `dplyr::filter()` in combination with the functions we learned for wrangling yesterday. If we have multiple sequential steps to perform, we can string them together using the *pipe operator* (`%>%`).

6.4.2 Merging data frames with `dplyr::*_join()`

Excel's `VLOOKUP` can also be used to merge data from separate tables or worksheets. Here, we'll use the `dplyr::*_join()` functions to merge separate data frames in R.

There are a number of ways to merge data frames in R. We'll use `full_join()`, `left_join()`, and `inner_join()` in this session.

From R Documentation (`?join`):

- `dplyr::full_join()`: “returns all rows and all columns from both x and y. Where there are not matching values, returns NA for the one missing.” Basically, nothing gets thrown out, even if a match doesn't exist - making `full_join()` the safest option for merging data frames. When in doubt, `full_join()`.
- `dplyr::left_join()`: “return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.”
- `dplyr::inner_join()`: “returns all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned.” This

will drop observations that don't have a match between the merged data frames, which makes it a riskier merging option if you're not sure what you're trying to do.

To clarify what the different joins are doing, let's first make a subset of the *fish_counts* data frame that only contains observations from 2016 and 2017.

```
fish_2016_2017 <- fish_counts %>%
  filter(year == 2016 | year == 2017)
```

Take a look to ensure that only those years are included with `View(fish_2016_2017)`. Now, let's merge it with our kelp fronds data in different ways.

6.4.2.1 `dplyr::full_join()` to merge data frames, keeping everything

When we join data frames in R, we need to tell R a couple of things (and it does the hard joining work for us):

- Which data frames we want to merge together
- Which variables* to merge by

Note: If there are **exactly matching** column names in the data frames you're merging, the `*_join()` functions will assume that you want to join by those columns. If there are *no* matching column names, you can specify which columns to join by manually. We'll do both here.

```
# Join the fish_counts and kelp_counts_abur together:
abur_kelp_join <- fish_2016_2017 %>%
  full_join(kelp_counts_abur) # Uh oh. An error message.
```

When we try to do that join, we get an error message: `Error: Can't join on 'year' x 'year' because of incompatible types (character / numeric)`

What's going on here? First, there's something fishy (ha) going on with the class of the *year* variable in *kelp_counts_abur*. Use the `class()` function to see how R understands that variable (remember, we use `$` to return a specific column from a data frame).

```
class(kelp_counts_abur$year)
```

```
## [1] "character"
```

So the variable is currently stored as a character. Why?

If we go back to the *kelp_counts_curated.xlsx* file, we'll see that the numbers in both the year and month column have been stored as *text*. There are several hints Excel gives us:

- Cells are left aligned, when values stored as numbers are right aligned
- The green triangles in the corner indicate some formatting

- The warning sign shows up when you click on one of the values with text formatting, and lets you know that the cell has been stored as text. We are given the option to reformat as numeric in Excel, but we'll do it here in R so we have a reproducible record of the change to the variable class.

There are a number of ways to do this in R. We'll use `dplyr::mutate()` to overwrite the existing `year` column while coercing it to class *numeric* using the `as.numeric()` function.

```
# Coerce the class of 'year' to numeric
kelp_counts_abur <- kelp_counts_abur %>%
  mutate(year = as.numeric(year))
```

Now if we check the class of the `year` variable in `kelp_counts_abur`, we'll see that it has been coerced to 'numeric':

```
class(kelp_counts_abur$year)
```

```
## [1] "numeric"
```

Question: Isn't it bad practice to overwrite variables, instead of just making a new one? Great question, and usually the answer is yes. Here, we feel fine with "overwriting" the `year` column because we're not changing anything about what's contained within the column, we're only changing how R understands it. Always use caution if overwriting variables, and if in doubt, add one instead!

OK, so now the class of `year` in the data frames we're joining is the same. Let's try that `full_join()` again:

TODO: consider using `fish_2016_2017` here to really show that `full_join()` keeps everything?

```
abur_kelp_join <- fish_2016_2017 %>%
  full_join(kelp_counts_abur)
```

```
## Joining, by = c("year", "site")
```

First, notice that R tells us which sites it is joining by - in this case, `year` and `site` since those were the two matching variables in both data frames.

Now look at the merged data frame with `View(abur_kelp_join)`. A few things to notice about how `full_join()` has worked:

1. All columns that existed in **both data frames** still exist.
2. All observations are retained, even if they don't have a match. In this case, notice that for other sites (not 'abur') the observation for fish still exists, even though there was no corresponding kelp data to merge with it. The kelp frond data from 2018 is also returned, even though the fish counts dataset did not have 'year == 2018' in it.

3. The kelp frond data is joined to *all observations* where the joining variables (*year*, *site*) are a match, which is why it is repeated 5 times for each year (once for each fish species).

Because all data (observations & columns) are retained, `full_join()` is the safest option if you're unclear about how to merge data frames.

6.4.2.2 `dplyr::left_join()` to merge data frames, keeping everything in the 'x' data frame and only matches from the 'y' data frame

TODO: think about a way to make `left_join()` clear. Need to join data frame 'y' that has something not contained in 'x', like a site or something, to show that it is dropped when left joining.

Now, we want to keep all observations in *fish_2016_2017*, and merge them with *kelp_counts_abur* while only keeping observations from *kelp_counts_abur* that match an observation within *fish_2016_2017*. So when we use a `left_join`, any information on kelp counts from 2018 should be dropped.

```
fish_kelp_2016_2017 <- fish_2016_2017 %>%
  left_join(kelp_counts_abur)
```

```
## Joining, by = c("year", "site")
```

6.4.2.3 `dplyr::inner_join()` to merge data frames, only keeping observations with a match in both

Use `inner_join()` if you know that you **only** want to retain observations when they match across both data frames. Caution: this is built to exclude any observations that don't match across data frames by joined variables - double check to make sure this is actually what you want to do!

For example, if we use `inner_join()` to merge *fish_counts* and *kelp_counts_abur*, then we are asking R to **only return observations where the joining variables (*year* and *site*) have matches in both data frames**. Let's see what the outcome is:

```
abur_kelp_inner_join <- fish_counts %>%
  inner_join(kelp_counts_abur)
```

```
## Joining, by = c("year", "site")
```

```
abur_kelp_inner_join
```

```
## # A tibble: 15 x 6
```

```
##   year site common_name total_count month total_fronds
##   <dbl> <chr> <chr>          <dbl> <chr>          <dbl>
```

##	1	2016	abur	black surfperch	2 7	307
##	2	2016	abur	blacksmith	1 7	307
##	3	2016	abur	garibaldi	1 7	307
##	4	2016	abur	rock wrasse	2 7	307
##	5	2016	abur	senorita	58 7	307
##	6	2017	abur	black surfperch	4 7	604
##	7	2017	abur	blacksmith	1 7	604
##	8	2017	abur	garibaldi	1 7	604
##	9	2017	abur	rock wrasse	57 7	604
##	10	2017	abur	senorita	64 7	604
##	11	2018	abur	black surfperch	1 7	3532
##	12	2018	abur	blacksmith	1 7	3532
##	13	2018	abur	garibaldi	1 7	3532
##	14	2018	abur	rock wrasse	1 7	3532
##	15	2018	abur	senorita	1 7	3532

Here, we see that only observations where there is a match for *year* and *site* are returned.

6.5 Fun facts

How is this similar to VLOOKUP in Excel? How does it differ?

From Microsoft Office Support, “use VLOOKUP when you need to find things in a table or a range by row.”

So, both `filter()` and VLOOKUP look through your data frame (or spreadsheet, in Excel) to look for observations that match your conditions. But they also differ in important ways:

- (1) By default VLOOKUP looks for and returns an observation for *approximate* matches (and you have to change the final argument to FALSE to look for an exact match). In contrast, by default `dplyr::filter()` will look for exact conditional matches.
- (2) VLOOKUP will look for and return information from the *first observation* that matches (or approximately matches) a condition. `dplyr::filter()` will return all observations (rows) that exactly match a condition.

6.6 Interludes (deep thoughts/openscapes)

- Idea: not overusing the pipe in really long sequences. What are other options? Why is that a concern? What are some ways to always know that what’s happening in a sequence is what you EXPECT is happening

in a sequence? tidylog, check intermediate data frames, sometimes write intermediate data frames, etc.

6.7 Our Turn Your Turn 1

6.8 Our Turn Your Turn 2

6.9 Efficiency Tips

Chapter 7

Tidying

7.1 Better practices [needs a better name]

How to be a nimble useR Modern useRs are nimble internet useRs something clever about cleaning I am the worst at naming things

7.2 Summary (a few sentences)

R ecosystem evolves and improves due to contributed work by the community, and this is a good thing. Being a nimble useR means being able to navigate/keep tabs on this ecosystem and find what you need. It also means working reproducibly, so you can re-run and update things more easily. Here we will teach you how to expect things and help yourself. Pay attention to urls.

7.3 Objectives (more detailed, bulletpoints?)

- expect there is a better way, how and where to look (20 mins)
 - CRAN
 - Twitter #rstats
 - rOpenSci
 - RStudio
 - Example: how to Google.
- hands-on with janitor (30+ mins)
 - discovery and quality assurance
 - installing from GitHub
 - big payoff for little effort

- hands-on with another excel-useful example: `skimr`?
- reproducibility (20 mins)
 - it's important, scripted

7.4 Resources

- Wilson et al. 2014 “Good enough practices”

7.5 Lessons teaching for each objective..... (objectives, examples)

7.5.1 Expect there's a better way chat

- give time for them to google?

7.5.2 Janitor

janitor & other things that will make your life easier with limited effort Janitor: up till now the column names have been fine. Until now.

7.5.2.1 Our turn your turn

Walk through and example and leave our code up, and have you do it but clean another dataset. Work with a neighbor.

7.5.3 Example: How to Google

Pay attention to URLs, build github/rmarkdown savviness (ex: `raw.githubusercontent.com`)

- I read this blog: <https://blog.revolutionanalytics.com/2018/08/how-to-use-r-with-excel.html>
- I've never heard of click on `openxlsx`, what is it
- Takes me here <https://www.rdocumentation.org/packages/openxlsx/versions/4.1.0.1>, but I want more info. How recently was it worked on? Does it interface with tidyverse? Click on “news”
- Takes me here. <https://raw.githubusercontent.com/awalker89/openxlsx/master/NEWS> . Not useful. But from this URL, - I see the username so I can edit this url to be <https://github.com/awalker89/openxlsx/>

7.6. *FUN FACTS (QUIRKY THINGS) - MAKING A NOTE OF THESE WHEREVER POSSIBLE FOR INTEREST*

- 1st thing: most recent commit was a year ago. Can poke around more, are there issues open, are they taken care of? Etc. I will probably not pursue using this right now. But good to have learned about it.

7.6 Fun facts (quirky things) - making a note of these wherever possible for interest (little “Did you know?” sections)

7.7 Interludes (deep thoughts/openscapes)

7.8 Our Turn Your Turn 2

7.9 Efficiency Tips

- browser efficiency tips
 - Rmd/github anchors for urls
 - press command to open a new tab

Reproducibility is important (this might be new to some people) Example: run everything start to finish and then closing it all and trying to do again In excel Vs R If your computer shuts off are you nervous to close it? Recreate it “What they didn’t forget to teach you about R” WTDF. uncool

Chapter 8

Formatting and Sharing

8.1 Summary (a few sentences)

We are going to focus on formatting a table nicely and sharing it by publishing it to GitHub.

8.2 Objectives (more detailed, bulletpoints?)

8.3 Resources

- Happy Git With R
- Openscapes lesson series

8.4 Introduction / our analytical plan

Generally, you can name things whatever you want; we try to do this in a way that is meaningful to you and others but is also succinct. But in this lesson there are going to be 2 things that must have specific names, which will let us create a website. These 2 things are `index.Rmd` and `gh-pages`. We'll see them soon.

<TODO: More here>

We'll create a new RMarkdown document — and this is the first of these 2 names we mentioned above. **Name your new R Markdown file `index.Rmd`.** Lowercase “i”.

TODO: read in data from googlesheets. Substrate cover?

Wrangle data

Now let's knit! Let's make this table look a bit nicer.

8.5 Formatting tables (15 minutes)

8.5.1 DT

TODO

8.5.2 kable and kablextra

TODO

8.6 Activity (5 min)

Do the following: TODO

1. wrangle xyz
2. something
3. knit to html

Great! Now we want to share this. We'll move beyond emailing Excel Files or sharing them via dropbox or google drive.

Let's talk about what GitHub is. You can sit back and close your laptops for a moment and we'll have a little tour.

8.7 GitHub for Publishing

GitHub is a powerful tool for collaborative coding with version control, but we are going to focus on some of its lesser-celebrated awesomeness: GitHub for publication.

We are going to work with GitHub from the browser only, because it makes the best use of our short time together. It is also a powerful way for folks to contribute and collaborate even if they are not involved in day-to-day hands-on analysis. So this might be good for you and your colleagues to have a gentle entryway into GitHub together.

GitHub can reduce friction collaboration: it gives us avenues for communicating and publishing methods, blogs, interactive graphics and more, without a lot of heavy lifting!

8.7.1 Issues

Every GitHub repository (shortened to “repo”) has a feature called Issues. Issues is GitHub’s project management and task-tracking feature.

Issues “track ideas, enhancements, tasks, or bugs for work on GitHub.” - GitHub

You create an Issue for a topic, and use it track progress or ask questions. You can provide links, describe updates, link to other Issues, and you can close the Issue when it is completed. You can also re-open previously-closed Issues.

Every GitHub repository has this Issues feature. This means that sometimes Issues are public and sometimes they are private.

- In a public repo, anyone with a GitHub username can create and comment on issues.
- In a private repo, only users with permission can create and comment on issues, or see them at all

We won’t talk about Issues more today in this course but if you’re interested read Jenny Bryan’s “Excuse Me, Do You Have a Moment to Talk About Version Control?” and Openscapes’ GitHub for Project Management chapter.

OK that’s all for the hands-off portion, let’s get back to our computers.

8.8 Create a GitHub user account

You will need to create **Github** account at <http://github.com>, if you don’t already have one. For username, I recommend all lower-case letters, short as you can.

Note! Shorter names that kind of identify you are better, and use your work email! (See Jenny Bryan’s username advice in Happy Git with R).

8.9 Create a repo

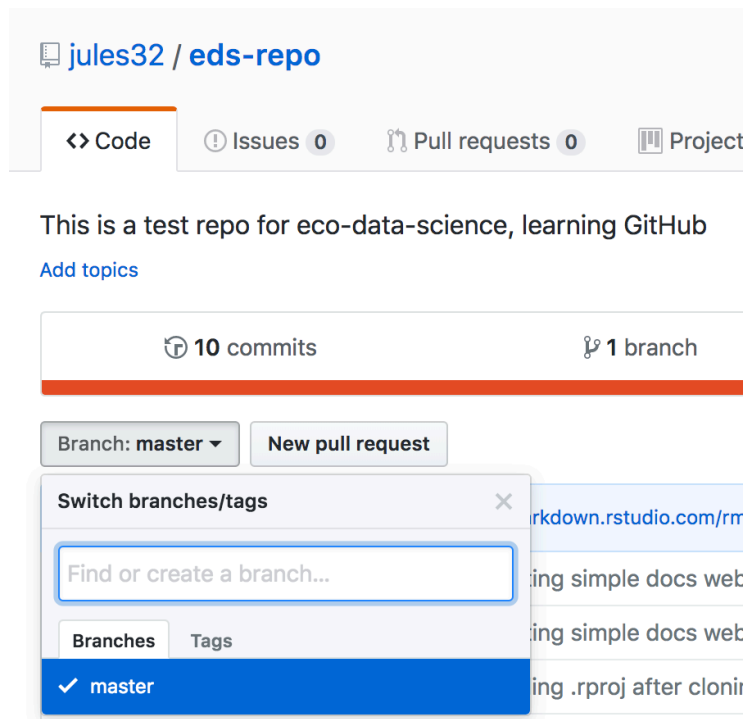
You can name it anything. Since it is going to be about our substrate analysis, I’ll name mine **substrate-report**.

8.9.1 Create a gh-pages branch

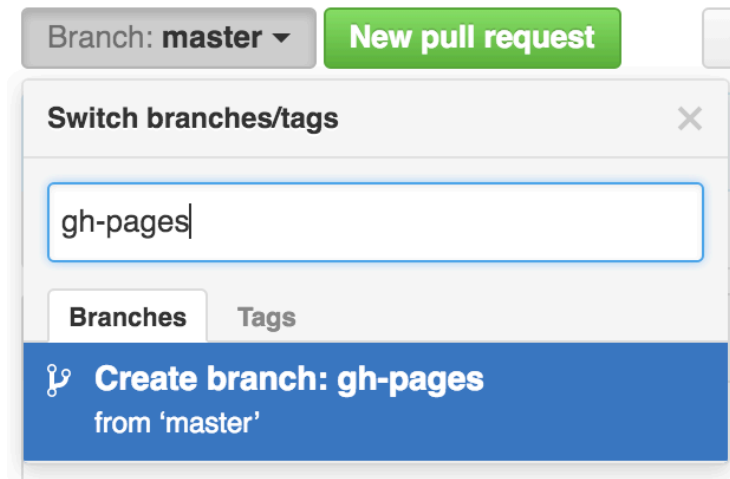
We’re not going to talk about branches and forks, except for one specific thing here. Branches and forks are a core feature of what makes GitHub super powerful for software development, and one of the first things you’ll see in GitHub tutorials geared towards software engineers. But I do not think that is the most relevant or smoothest entryway for those of us who are scientists fairly new to collaborative coding and version control.

That said, by default, when you create a new repo you begin with one branch, and it is named `master`. When you create new branches, you can name them whatever you want. However, if you name one `gh-pages` (all lowercase, with a `-` and no spaces), this will let you create a website. And that’s our plan. Everyone, do the following to create a `gh-pages` branch:

On the homepage for your repo on GitHub.com, click the button that says “Branch:master”. Here, you can switch to another branch (right now there aren’t any others besides `master`), or create one by typing a new name.

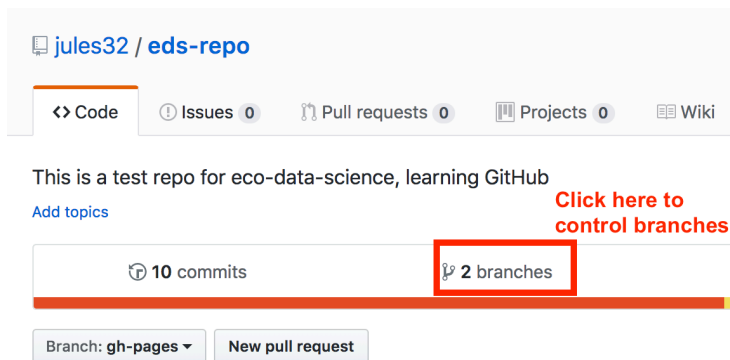


Let’s type `gh-pages`.

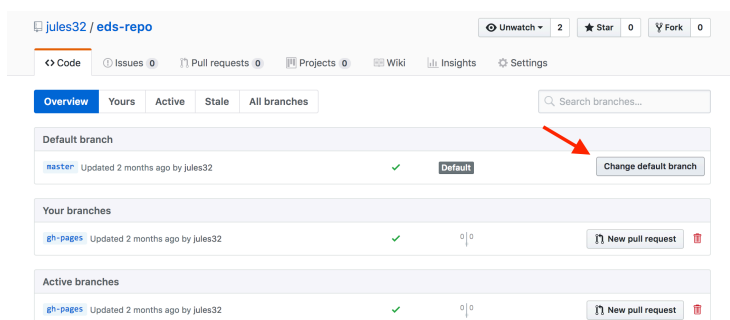


Let's also change **gh-pages** to the default branch and delete the master branch: this will be a one-time-only thing that we do here:

First click to control branches:



And then click to change the default branch to **gh-pages**. I like to then delete the **master** branch when it has the little red trash can next to it. It will make you confirm that you really want to delete it, which I do!



8.10 Drag and Drop

Drag and drop your Rmd and html files.

Next, you'll need to commit to uploading them by writing a commit message. GitHub provides a suggested default message, but get into the habit of writing an actual message to yourself/others (breadcrumbs). Great! Now your files are listed as files in the repository.

(These are static; if you make a change to the Rmd it won't reflect on the html because you'd need to knit it again.)

8.11 Visit your website!

Go to *YOUR-USER-NAME.github.io/REPO-NAME*

Trouble?

- 404 error? Remove trailing / from the url
- didn't name your file `index.Rmd`? Go to *YOUR-USER-NAME.github.io/REPO-NAME/FILENAME*
- TODO? Remove trailing .Rmd from the url

8.12 Edit the README

The last thing we'll do here is to update our README file. READMEs are important because they are essentially a home page for your project, and GitHub displays them nicely and GitHub users expect to see them with high-level information about your project.

And, they should be useful for you and your collaborators, whether that is "future you" or "future us".

First a Disclaimer: you don't want to edit from the browser for most things – you would want to "clone" the repo to your local computer and leverage more goodies & power. However, you will sometimes edit in the browser, and it's a good entry point for us today, and maybe for onboarding folks in your lab in the future.

Why not edit in the browser? You don't want to overwrite each other or forget yourself. Good for quick md editing, not script editing.

So let's click on the pencil and write a bit of information about our project. I'm going to write:

`A short exploration of substrate data from the California LTER.`

[Rendered html file](my-url). TODO

This is great because now you don't need to remember everything about what we did with the DT package if you can remember your GitHub username and how to navigate to this repo. And you can share this URL with others! Hooray!

8.13 Activity (5 min)

- Add your name to the README and commit it.
- MORE

8.14 recap

Intentionally “lightweight” entry to GitHub. But more powerful than this. We wanted to give you a taste of what is possible with GitHub.

To take your analyses to the next level and use GitHub with R for version control, read Jenny Bryan's Happy Git with R when you're home.

8.15 Interludes (deep thoughts/openscapes)

8.16 Efficiency Tips

Chapter 9

Synthesis

- 9.1 Summary (a few sentences)
- 9.2 Objectives (more detailed, bulletpoints?)
- 9.3 Resources
- 9.4 Lessons teaching for each objective..... (objectives, examples)
- 9.5 Fun facts (quirky things) - making a note of these wherever possible for interest (little “Did you know?” sections)
- 9.6 Interludes (deep thoughts/openscapes)
- 9.7 Our Turn Your Turn 1
- 9.8 Our Turn Your Turn 2
- 9.9 Efficiency Tips