

R for Excel Users

Julie Lowndes & Allison Horst

2019-12-01

Contents

1	Welcome	7
1.1	Prerequisites	8
2	Overview	9
2.1	Summary	9
2.2	Objectives (more detailed, bulletpoints?)	9
2.3	Resources	9
2.4	Overview	10
2.5	If R were an Airplane	12
2.6	RStudio Orientation	12
2.7	Deep thought	16
2.8	Don't save the workspace	17
2.9	Deep thought: keep the raw data raw.	17
2.10	Activity 1	17
2.11	Activity 2	17
2.12	Efficiency Tips	17
2.13	Troubleshooting	17
3	R & RStudio, RMarkdown	19
3.1	Summary	19
3.2	Objectives (more detailed, bulletpoints?)	19
3.3	Resources	19
3.4	Intro to RMarkdown	19
3.5	Activity	24
3.6	Deep thoughts	24
3.7	Fun facts (quirky things) - making a note of these wherever possible for interest (little "Did you know?" sections)	24
3.8	Interludes (deep thoughts/openscapes)	24
3.9	Interludes (deep thoughts/openscapes)	24
3.10	Efficiency Tips	24
4	GitHub	25
4.1	Fun facts (quirky things) - making a note of these wherever possible for interest (little "Did you know?" sections)	25

4.2	Interludes (deep thoughts/openscapes)	25
4.3	Our Turn Your Turn 1	25
4.4	Our Turn Your Turn 2	25
4.5	Efficiency Tips	25
5	readxl	27
5.1	Summary	27
5.2	Objectives	27
5.3	Resources	27
5.4	Lesson	28
5.5	Fun facts ideas:	35
5.6	Interludes (deep thoughts/openscapes)	35
5.7	Activity: Import some invertebrates!	35
5.8	Efficiency Tips	36
5.9	Deep thoughts	36
6	Graphs with ggplot2	37
6.1	Summary	37
6.2	Objectives	37
6.3	Resources	38
7	Visualizing: ggplot2	39
7.1	Objectives & Resources	39
7.2	Install our first package: tidyverse	40
7.3	Load data	41
7.4	Plotting with ggplot2	42
7.5	Data	43
7.6	Building your plots iteratively	45
7.7	Customizing plots	47
7.8	ggplot2 themes	47
7.9	Faceting	49
7.10	Geometric objects (geoms)	51
7.11	Bar charts	55
7.12	Arranging and exporting plots	59
7.13	Bonus	60
7.14	Save and push to GitHub	60
7.15	Fun facts (quirky things) - making a note of these wherever possible for interest (little “Did you know?” sections)	60
7.16	Interludes (deep thoughts/openscapes)	60
7.17	Our Turn Your Turn 1	60
7.18	Our Turn Your Turn 2	60
7.19	Efficiency Tips	60
8	dplyr and Pivot Tables	61
8.1	Summary (a few sentences)	61
8.2	Objectives (more detailed, bulletpoints?)	61

8.3	Resources	62
8.4	Lesson	62
8.5	dplyr overview	64
8.6	<code>group_by()</code> <code>%>% summarize()</code>	65
8.7	<code>mutate()</code>	68
8.8	<code>mutate()</code> vs <code>summarize()</code>	70
8.9	Deep thoughts	71
8.10	Efficiency Tips	71
9	Tidying	73
9.1	Summary	73
9.2	Objectives	73
9.3	Resources	73
9.4	Lesson	74
10	Dplyr and vlookups	87
10.1	Summary	87
10.2	Objectives	87
10.3	Resources	88
10.4	Lessons	88
10.5	Fun / kind of scary facts	100
10.6	Interludes (deep thoughts/openscapes)	100
10.7	Efficiency Tips	100
11	Synthesis	101
11.1	Summary	101
11.2	Objectives	101
11.3	Resources	102
11.4	Lesson	102
11.5	Fun facts (quirky things) - making a note of these wherever possible for interest (little “Did you know?” sections)	104
11.6	Interludes (deep thoughts/openscapes)	104
11.7	Efficiency Tips	104

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

TODO: shorten so it's just the overview (ie cut the Rscripts)

This 2-day workshop aims to introduce you to R and help you develop good habits and workflows. Coming from Excel.

Excel is great for data entry. Can also be good for looking at data and feeling like you can touch it. But this can also get problematic with extending to analyses. Keep raw data raw, TODO

2.1 Summary

Welcome! We'll introduce the workshop and the RStudio interface (IDE).

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.

Excel is great for a lot of things:

- data entry
- quick data exploration and reactive plots

Not great for a lot of things:

- What did I do?
- That terrible feeling

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...

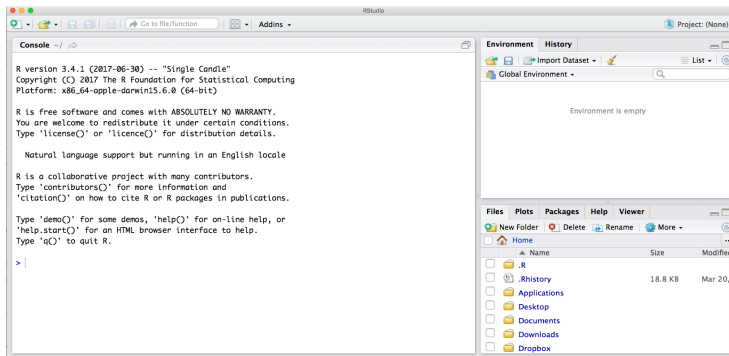
By the end of this course you'll produce this report that you can reproduce, which means... Introduce the problem we will solve. Eg: (just an idea maybe time-series is not a great idea) SMALL PROBLEM. (4 mins) Show data files, We will discuss our analysis plan (only enough to motivate!) Create a report,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

2.5 If R were an Airplane

2.6 RStudio Orientation

Open RStudio for the first time.

Launch RStudio/R.



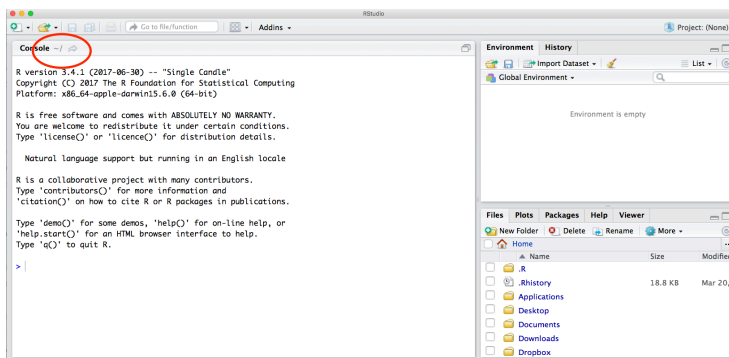
Notice the default panes:

- 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: `~/`.



2.6.1 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)
```

2.7 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: $=A1 * 2$, $=A2 * 2$, ..., $=An * 2$.

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.

—>

2.7.1 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

R & RStudio, RMarkdown

TODO: Intro: we're going to start off with RMarkdown.

TODO: break into “R chunks” and “Markdown sections”

TODO: harmonize with OHI's training

Introduce projects and setup data/ folder

TODO (write out): Intro, knitting 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.

3.1 Summary

We'll learn RMarkdown, which helps you tell a story with your data analysis because you can write text alongside the code. We are actually learning two languages at once: R and Markdown.

3.2 Objectives (more detailed, bulletpoints?)

3.3 Resources

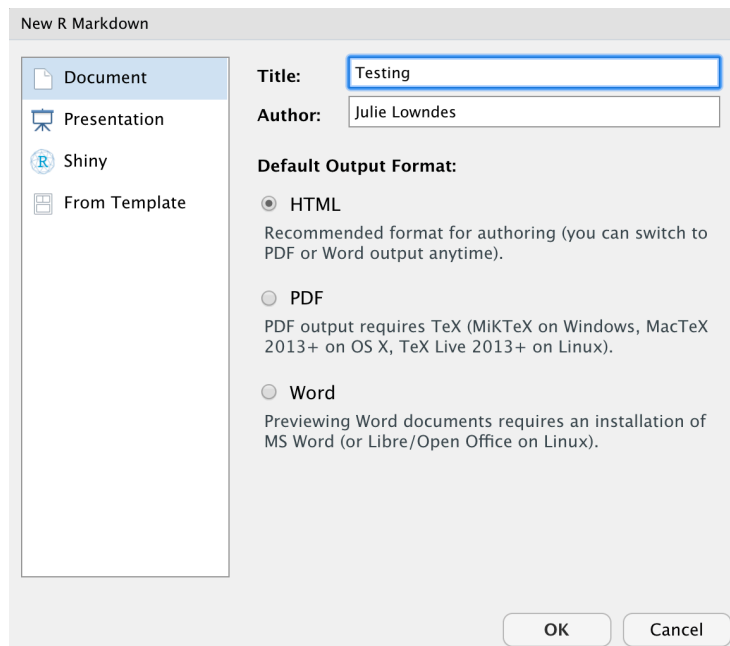
3.4 Intro to RMarkdown

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. This is super cool, and

really best to experience before we talk about it. So let's do this together.

3.4.1 Create an RMarkdown file

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



You can give it a Title like “Testing” (a name that’s totally ok to use as you’re trying something out). 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:

- Title and Author are auto-filled, and the today’s date has been added
- There are white and grey sections. These are the 2 main languages that make up an RMarkdown file.
 - **Grey sections are R code**
 - **White sections are Markdown text**

```

1 ---
2 title: "Testing"
3 author: "Julie Lowndes"
4 date: "11/14/2019"
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 documents.
15 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 output of any
18 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 generated the
33 plot.

```

3.4.2 Knit your RMarkdown file

Let's go ahead and "Knit" by clicking the blue yarn at the top of the RMarkdown file. It's going to ask us to save first, I'll name mine "testing.Rmd".

How cool is this, we've just made an html file, a webpage that we are viewing locally on our own computers. Knitting this RMarkdown document has rendered — we also say formatted — both the Markdown text (white) and the R code (grey), and the it also executed — we also say ran — the R code.

Let's have a look at them side-by-side:

The screenshot displays the RStudio interface with two panes. The left pane shows the R Markdown source file 'testing.Rmd' with line numbers 1 through 33. The right pane shows the rendered HTML output. The HTML output includes the title 'Testing', author 'Julie Lowndes', and date '11/14/2019'. It also contains the 'R Markdown' section with a brief explanation of the format. The 'Including Plots' section shows the output of the R code chunks: a summary of the 'cars' dataset and a scatter plot of 'pressure' vs 'temperature'.

	mpg	displ	wt	qsec	vs	am	gear	carb
##	16.99	161.9	3513	18.71	0	1	4	4
##	17.01	161.9	3513	18.71	0	1	4	4
##	17.02	161.9	3513	18.71	0	1	4	4
##	17.03	161.9	3513	18.71	0	1	4	4
##	17.04	161.9	3513	18.71	0	1	4	4
##	17.05	161.9	3513	18.71	0	1	4	4
##	17.06	161.9	3513	18.71	0	1	4	4
##	17.07	161.9	3513	18.71	0	1	4	4
##	17.08	161.9	3513	18.71	0	1	4	4
##	17.09	161.9	3513	18.71	0	1	4	4
##	17.10	161.9	3513	18.71	0	1	4	4
##	17.11	161.9	3513	18.71	0	1	4	4
##	17.12	161.9	3513	18.71	0	1	4	4
##	17.13	161.9	3513	18.71	0	1	4	4
##	17.14	161.9	3513	18.71	0	1	4	4
##	17.15	161.9	3513	18.71	0	1	4	4
##	17.16	161.9	3513	18.71	0	1	4	4
##	17.17	161.9	3513	18.71	0	1	4	4
##	17.18	161.9	3513	18.71	0	1	4	4
##	17.19	161.9	3513	18.71	0	1	4	4
##	17.20	161.9	3513	18.71	0	1	4	4
##	17.21	161.9	3513	18.71	0	1	4	4
##	17.22	161.9	3513	18.71	0	1	4	4
##	17.23	161.9	3513	18.71	0	1	4	4
##	17.24	161.9	3513	18.71	0	1	4	4
##	17.25	161.9	3513	18.71	0	1	4	4
##	17.26	161.9	3513	18.71	0	1	4	4
##	17.27	161.9	3513	18.71	0	1	4	4
##	17.28	161.9	3513	18.71	0	1	4	4
##	17.29	161.9	3513	18.71	0	1	4	4
##	17.30	161.9	3513	18.71	0	1	4	4
##	17.31	161.9	3513	18.71	0	1	4	4
##	17.32	161.9	3513	18.71	0	1	4	4
##	17.33	161.9	3513	18.71	0	1	4	4
##	17.34	161.9	3513	18.71	0	1	4	4
##	17.35	161.9	3513	18.71	0	1	4	4
##	17.36	161.9	3513	18.71	0	1	4	4
##	17.37	161.9	3513	18.71	0	1	4	4
##	17.38	161.9	3513	18.71	0	1	4	4
##	17.39	161.9	3513	18.71	0	1	4	4
##	17.40	161.9	3513	18.71	0	1	4	4
##	17.41	161.9	3513	18.71	0	1	4	4
##	17.42	161.9	3513	18.71	0	1	4	4
##	17.43	161.9	3513	18.71	0	1	4	4
##	17.44	161.9	3513	18.71	0	1	4	4
##	17.45	161.9	3513	18.71	0	1	4	4
##	17.46	161.9	3513	18.71	0	1	4	4
##	17.47	161.9	3513	18.71	0	1	4	4
##	17.48	161.9	3513	18.71	0	1	4	4
##	17.49	161.9	3513	18.71	0	1	4	4
##	17.50	161.9	3513	18.71	0	1	4	4
##	17.51	161.9	3513	18.71	0	1	4	4
##	17.52	161.9	3513	18.71	0	1	4	4
##	17.53	161.9	3513	18.71	0	1	4	4
##	17.54	161.9	3513	18.71	0	1	4	4
##	17.55	161.9	3513	18.71	0	1	4	4
##	17.56	161.9	3513	18.71	0	1	4	4
##	17.57	161.9	3513	18.71	0	1	4	4
##	17.58	161.9	3513	18.71	0	1	4	4
##	17.59	161.9	3513	18.71	0	1	4	4
##	17.60	161.9	3513	18.71	0	1	4	4
##	17.61	161.9	3513	18.71	0	1	4	4
##	17.62	161.9	3513	18.71	0	1	4	4
##	17.63	161.9	3513	18.71	0	1	4	4
##	17.64	161.9	3513	18.71	0	1	4	4
##	17.65	161.9	3513	18.71	0	1	4	4
##	17.66	161.9	3513	18.71	0	1	4	4
##	17.67	161.9	3513	18.71	0	1	4	4
##	17.68	161.9	3513	18.71	0	1	4	4
##	17.69	161.9	3513	18.71	0	1	4	4
##	17.70	161.9	3513	18.71	0	1	4	4
##	17.71	161.9	3513	18.71	0	1	4	4
##	17.72	161.9	3513	18.71	0	1	4	4
##	17.73	161.9	3513	18.71	0	1	4	4
##	17.74	161.9	3513	18.71	0	1	4	4
##	17.75	161.9	3513	18.71	0	1	4	4
##	17.76	161.9	3513	18.71	0	1	4	4
##	17.77	161.9	3513	18.71	0	1	4	4
##	17.78	161.9	3513	18.71	0	1	4	4
##	17.79	161.9	3513	18.71	0	1	4	4
##	17.80	161.9	3513	18.71	0	1	4	4
##	17.81	161.9	3513	18.71	0	1	4	4
##	17.82	161.9	3513	18.71	0	1	4	4
##	17.83	161.9	3513	18.71	0	1	4	4
##	17.84	161.9	3513	18.71	0	1	4	4
##	17.85	161.9	3513	18.71	0	1	4	4
##	17.86	161.9	3513	18.71	0	1	4	4
##	17.87	161.9	3513	18.71	0	1	4	4
##	17.88	161.9	3513	18.71	0	1	4	4
##	17.89	161.9	3513	18.71	0	1	4	4
##	17.90	161.9	3513	18.71	0	1	4	4
##	17.91	161.9	3513	18.71	0	1	4	4
##	17.92	161.9	3513	18.71	0	1	4	4
##	17.93	161.9	3513	18.71	0	1	4	4
##	17.94	161.9	3513	18.71	0	1	4	4
##	17.95	161.9	3513	18.71	0	1	4	4
##	17.96	161.9	3513	18.71	0	1	4	4
##	17.97	161.9	3513	18.71	0	1	4	4
##	17.98	161.9	3513	18.71	0	1	4	4
##	17.99	161.9	3513	18.71	0	1	4	4
##	18.00	161.9	3513	18.71	0	1	4	4

Let's take a deeper look at these two files. So much of learning to code is looking for patterns.

3.4.3 Activity (5 mins)

Look at these two files while talking with your neighbor: What do you notice between the two files? Start off with these observations:

- Markdown text (white sections): The two hashtags **##** cause the following text to be displayed as a header: the text is larger and in bold
- R code (grey sections): The `summary()` function outputs summary statistics and the `plot()` function creates a plot!

And now let's talk about this.

3.4.4 R code chunks

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`. COME BACK!!!

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/>

3.4.5 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.

3.4.6 R 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

Cool tip: doesn't have to be only R, other languages supported.

3.4.7 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.

3.5 Activity

1. knit!

Delete everything! Just by being an .Rmd file, this will knit

3.6 Deep thoughts

Command-Z is the best

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

3.8 Interludes (deep thoughts/openscapes)

3.9 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.

3.9.1 RMarkdown video (1-minute)

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

What is RMarkdown?

3.10 Efficiency Tips

Chapter 4

GitHub

Setup GitHub GitHub Scrap plans for publishing and issues: setup version control and practice like OHI data-science-training. Practice committing the README and .gitignore Copy our rproj into this repo and push

Add at the end: create an Rmd for our next lesson after lunch? Yes, and add a header with their inform

Create the github repo: ‘r-and-excel’

4.0.1 Reading

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

4.2 Interludes (deep thoughts/openscapes)

4.3 Our Turn Your Turn 1

4.4 Our Turn Your Turn 2

4.5 Efficiency Tips

Chapter 5

readxl

5.1 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.

5.2 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_xlsx()`
- Workflows with **readxl**: considerations, limitations, reproducibility

5.3 Resources

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

- readxl Workflows article (from tidyverse.org)

5.4 Lesson

5.4.1 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 TODO: XX files for this workshop:

- fish_counts_curated.csv
- invert_counts_curated.xlsx
- kelp_counts_curated.xlsx
- substrate_cover_curated.xlsx
- lobster_counts.csv TODO: add this?
- national_parks.xlsx TODO: add this?

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.

5.4.2 In the .Rmd you just created, attach the tidyverse, readxl and writexl packages

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 packages within the set-up chunk in R Markdown, so we’ll also attach the `readxl` and `writexl` packages there.

Here’s our first code chunk:

```
{r setup, eval = FALSE}
knitr::opts_chunk$set(echo = TRUE)

# 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.

5.4.3 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 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.

5.4.4 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> <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!

5.4.5 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.

5.4.6 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_fronde
##   <chr> <chr> <chr>      <dbl>
## 1 2016   7    golb         2557
## 2 2017   7    golb         1575
## 3 2018   7    golb         1629
```

5.4.7 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: TODO 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
```

5.4.8 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::write_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")
```

5.5 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`?

5.6 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)

5.7 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

TODO: Make these activities more interesting

- 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"

```
# 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()

# Writing a csv "inverts_july.csv":
write_csv(inverts_july, "inverts_july.csv")
```

5.8 Efficiency Tips

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

5.9 Deep thoughts

- Economist article about gene > dates issue in Excel
- Mine: data frame of bike casualties in NC, column names are age ranges but some of them import as dates
- Excel makes some wrong assumptions and doesn't give you a heads up about its decision making

Chapter 6

Graphs with ggplot2

TODO: Allison, smooth this over and connect Channel Islands National Park to LTER SBC datasets

6.1 Summary

Now that we know how to *get* some data, the next thing we'll probably want to do is *look* at it. In Excel, graphs are made by manually selecting options - which, as we've discussed previously, may not be the best option for reproducibility.

Using `ggplot2`, the graphics package within the `tidyverse`, we'll write reproducible code to manually and thoughtfully build our graphs.

“ggplot2 implements the grammar of graphics, a coherent system for describing and building graphs. With ggplot2, you can do more faster by learning one system and applying it in many places.” - R4DS

6.2 Objectives

- Read in National Parks visitation data from a multisheet Excel workbook
- Explore visitation for Channel Islands National Park, and see how that compares to all parks
- Customize a gg-graph
- Export and save your graphs

TODO: Get CINP and all NPs data into a multisheet Excel workbook, then continue

6.3 Resources

Chapter 7

Visualizing: ggplot2

Why do we start with data visualization? Not only is data viz a big part of analysis, it's a way to SEE your progress as you learn to code.

7.1 Objectives & Resources

7.1.1 Objectives

- install our first package, **ggplot2**, by installing **tidyverse**
- learn ggplot2 with a national parks visitation dataset (important to play with other data than your own, you'll learn something.)
- practice writing a script in RMarkdown
- practice the rstudio-github workflow

7.1.2 Resources

Here are some additional resources for data visualization in R:

- [ggplot2-cheatsheet-2.1.pdf](#)
- Interactive Plots and Maps - Environmental Informatics
- Graphs with ggplot2 - Cookbook for R
- ggplot2 Essentials - STHDA
- “Why I use ggplot2” - David Robinson Blog Post
- Melanie Frazier’s GitHub Lesson at University of Queensland

7.2 Install our first package: tidyverse

Packages are bundles of functions, along with help pages and other goodies that make them easier for others to use, (ie. vignettes).

So far we've been using packages that are already included in *base R*. These can be considered *out-of-the-box* packages and include things such as `sum` and `mean`. You can also download and install packages created by the vast and growing R user community. The most traditional place to download packages is from CRAN, the Comprehensive R Archive Network. This is where you went to download R originally, and will go again to look for updates. You can also install packages directly from GitHub, which we'll do tomorrow.

You don't need to go to CRAN's website to install packages, we can do it from within R with the command `install.packages("package-name-in-quotes")`.

We are going to be using the package `ggplot2`, which is actually bundled into a huge package called `tidyverse`. We will install `tidyverse` now, and use a few functions from the packages within. Also, check out tidyverse.org/.

```
## from CRAN:
```

```
install.packages("tidyverse") ## do this once only to install the package on your comp
```

```
library(tidyverse) ## do this every time you restart R and need it
```

When you do this, it will tell you which packages are inside of `tidyverse` that have also been installed. Note that there are a few name conflicts; it is alerting you that we'll be using two functions from `dplyr` instead of the built-in `stats` package.

What's the difference between `install.packages()` and `library()`? Why do you need both? Here's an analogy:

- `install.packages()` is setting up electricity for your house. Just need to do this once (let's ignore monthly bills).
- `library()` is turning on the lights. You only turn them on when you need them, otherwise it wouldn't be efficient. And when you quit R, it turns the lights off, but the electricity lines are still there. So when you come back, you'll have to turn them on again with `library()`, but you already have your electricity set up.

You can also install packages by going to the Packages tab in the bottom right pane. You can see the packages that you have installed (listed) and loaded (checkbox). You can also install packages using the install button, or check to see if any of your installed packages have updates available (update button). You can also click on the name of the package to see all the functions inside it — this is a super helpful feature that I use all the time.

7.3 Load data

Copy and paste the code chunk below and read it in to your RStudio to load the five datasets we will use in this section.

```
#National Parks in California
ca <- read_csv("https://raw.githubusercontent.com/OHI-Science/data-science-training/master/data/c

## Parsed with column specification:
## cols(
##   region = col_character(),
##   state = col_character(),
##   code = col_character(),
##   park_name = col_character(),
##   type = col_character(),
##   visitors = col_double(),
##   year = col_double()
## )

#Acadia National Park
acadia <- read_csv("https://raw.githubusercontent.com/OHI-Science/data-science-training/master/d

## Parsed with column specification:
## cols(
##   region = col_character(),
##   state = col_character(),
##   code = col_character(),
##   park_name = col_character(),
##   type = col_character(),
##   visitors = col_double(),
##   year = col_double()
## )

#Southeast US National Parks
se <- read_csv("https://raw.githubusercontent.com/OHI-Science/data-science-training/master/data/s

## Parsed with column specification:
## cols(
##   region = col_character(),
##   state = col_character(),
##   code = col_character(),
##   park_name = col_character(),
##   type = col_character(),
##   visitors = col_double(),
##   year = col_double()
## )
```

```

#2016 Visitation for all Pacific West National Parks
visit_16 <- read_csv("https://raw.githubusercontent.com/OHI-Science/data-science-training/1

## Parsed with column specification:
## cols(
##   region = col_character(),
##   state = col_character(),
##   code = col_character(),
##   park_name = col_character(),
##   type = col_character(),
##   visitors = col_double(),
##   year = col_double()
## )

#All Nationally designated sites in Massachusetts
mass <- read_csv("https://raw.githubusercontent.com/OHI-Science/data-science-training/1

## Parsed with column specification:
## cols(
##   region = col_character(),
##   state = col_character(),
##   code = col_character(),
##   park_name = col_character(),
##   type = col_character(),
##   visitors = col_double(),
##   year = col_double()
## )

```

7.4 Plotting with ggplot2

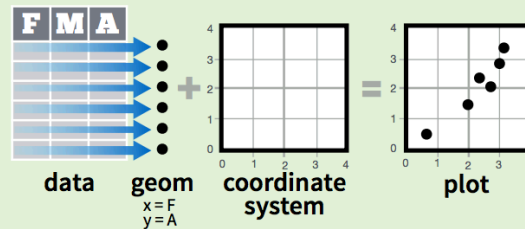
ggplot2 is a plotting package that makes it simple to create complex plots from data in a data frame. It provides a more programmatic interface for specifying what variables to plot, how they are displayed, and general visual properties. Therefore, we only need minimal changes if the underlying data change or if we decide to change from a bar plot to a scatterplot. This helps in creating publication quality plots with minimal amounts of adjustments and tweaking.

ggplot likes data in the ‘long’ format: i.e., a column for every dimension, and a row for every observation. Well structured data will save you lots of time when making figures with ggplot.

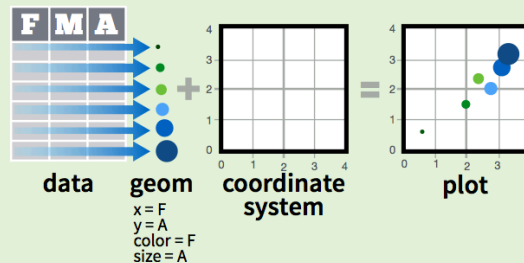
ggplot graphics are built step by step by adding new elements. Adding layers in this fashion allows for extensive flexibility and customization of plots.

Basics

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same few components: a **data** set, a set of **geoms**—visual marks that represent data points, and a **coordinate system**.



To display data values, map variables in the data set to aesthetic properties of the geom like **size**, **color**, and **x** and **y** locations.



7.5 Data

We are going to use a National Park visitation dataset (from the National Park Service at <https://irma.nps.gov/Stats/SSRSReports>). Read in the data using `read_csv` and take a look at the first few rows using `head()` or `View()`.

```
head(ca)
```

```
## # A tibble: 6 x 7
##   region state code  park_name          type      visitors  year
##   <chr>  <chr> <chr> <chr>          <chr>    <dbl> <dbl>
## 1 PW    CA    CHIS Channel Islands National P~ National P~    1200 1963
## 2 PW    CA    CHIS Channel Islands National P~ National P~    1500 1964
```

##	3	PW	CA	CHIS	Channel Islands National P~	National P~	1600	1965
##	4	PW	CA	CHIS	Channel Islands National P~	National P~	300	1966
##	5	PW	CA	CHIS	Channel Islands National P~	National P~	15700	1967
##	6	PW	CA	CHIS	Channel Islands National P~	National P~	31000	1968

This dataframe is already in a *long* format where all rows are an observation and all columns are variables. Among the variables in `ca` are:

1. `region`, US region where park is located.
2. `visitors`, the annual visitation for each year

To build a ggplot, we need to:

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

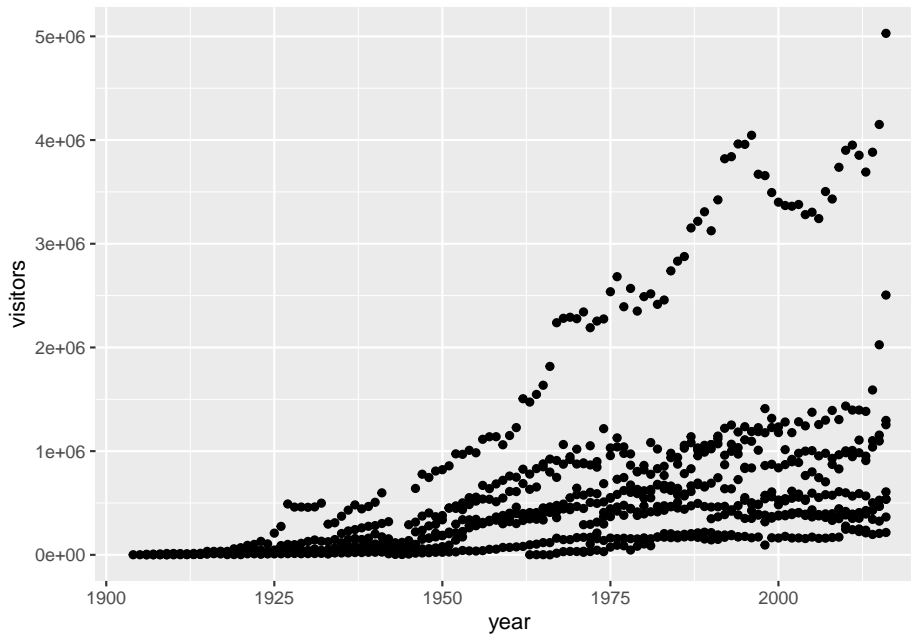
```
ggplot(data = ca)
```

- add `geoms` – graphical representation of the data in the plot (points, lines, bars). `ggplot2` offers many different geoms; we will use some common ones today, including: * `geom_point()` for scatter plots, dot plots, etc. * `geom_bar()` for bar charts * `geom_line()` for trend lines, time-series, etc.

To add a geom to the plot use `+` operator. Because we have two continuous variables,

let's use `geom_point()` first and then assign x and y aesthetics (`aes`):

```
ggplot(data = ca) +  
  geom_point(aes(x = year, y = visitors))
```



Notes:

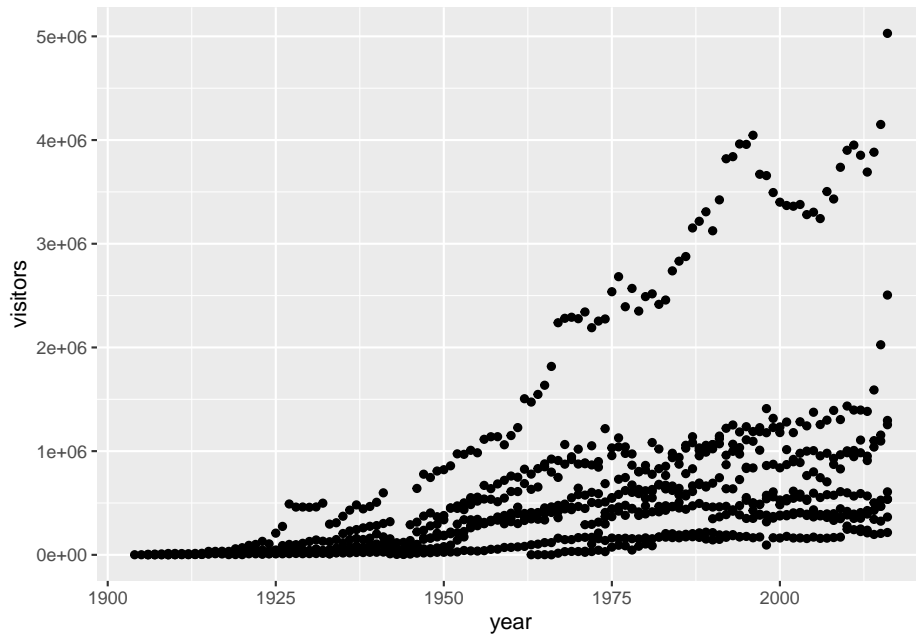
- Anything you put in the `ggplot()` function can be seen by any geom layers that you add (i.e., these are universal plot settings). This includes the x and y axis you set up in `aes()`.
- You can also specify aesthetics for a given geom independently of the aesthetics defined globally in the `ggplot()` function.
- The `+` sign used to add layers must be placed at the end of each line containing a layer. If, instead, the `+` sign is added in the line before the other layer, `ggplot2` will not add the new layer and will return an error message.

STOP: let's Commit, Pull and Push to GitHub

7.6 Building your plots iteratively

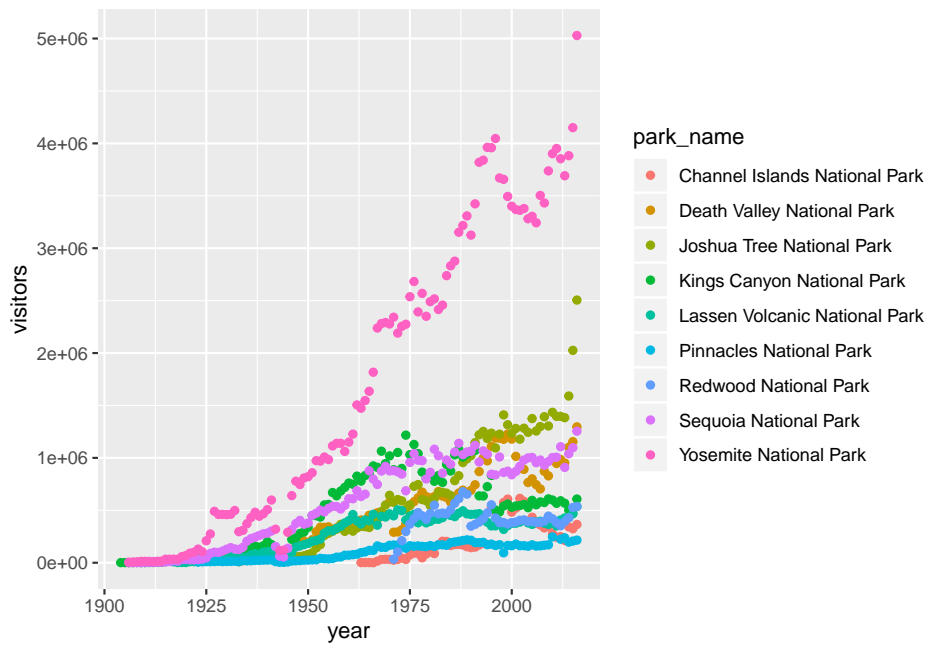
Building plots with `ggplot` is typically an iterative process. We start by defining the dataset we'll use, lay the axes, and choose a geom:

```
ggplot(data = ca) +  
  geom_point(aes(x = year, y = visitors))
```



This isn't necessarily a useful way to look at the data. We can distinguish each park by added the `color` argument to the `aes`:

```
ggplot(data = ca) +  
  geom_point(aes(x = year, y = visitors, color = park_name))
```

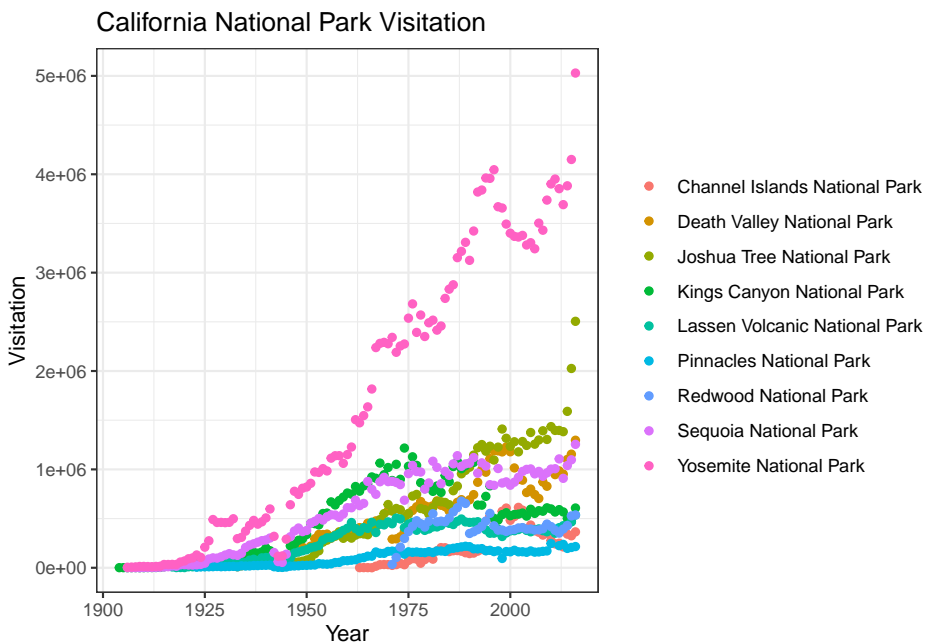


7.7 Customizing plots

Take a look at the `ggplot2` cheat sheet, and think of ways you could improve the plot.

Now, let's capitalize the x and y axis labels and add a main title to the figure. I also like to remove that standard gray background using a different `theme`. Many themes come built into the `ggplot2` package. My preference is `theme_bw()` but once you start typing `theme_` a list of options will pop up. The last thing I'm going to do is remove the legend title.

```
ggplot(data = ca) +
  geom_point(aes(x = year, y = visitors, color = park_name)) +
  labs(x = "Year",
       y = "Visitation",
       title = "California National Park Visitation") +
  theme_bw() +
  theme(legend.title=element_blank())
```



7.8 ggplot2 themes

In addition to `theme_bw()`, which changes the plot background to white, `ggplot2` comes with several other themes which can be useful to quickly change the look of your visualization.

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

7.8.1 Exercise (10 min)

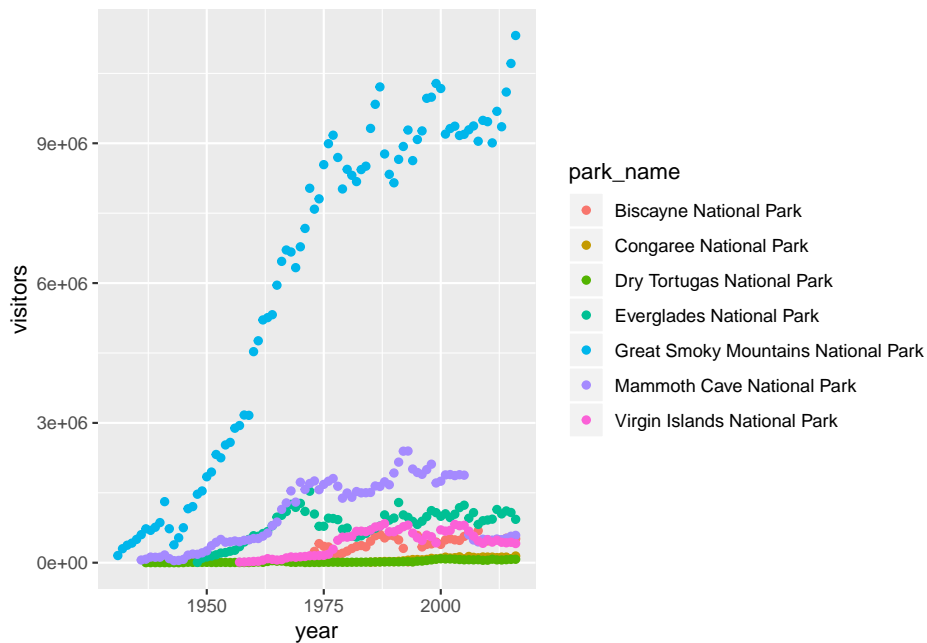
1. Using the `se` dataset, make a scatterplot showing visitation to all national parks in the Southeast region with color identifying individual parks.
2. Change the plot so that color indicates `state`.
3. Customize by adding your own title and theme. You can also change the text sizes and angles. Try applying a 45 degree angle to the x-axis. Use your cheatsheet!
4. In the code below, why isn't the data showing up?

```
ggplot(data = se, aes(x = year, y = visitors))
```

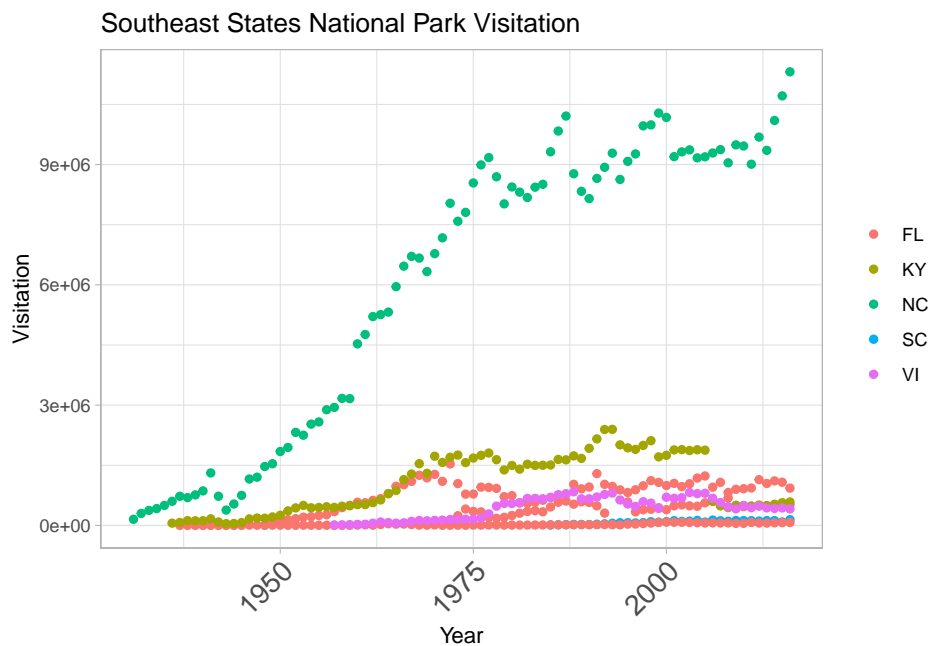
7.8.1.0.1 Answers (no peeking)

1.

```
ggplot(data = se) +  
  geom_point(aes(x = year, y = visitors, color = park_name))
```




```
# 2. & 3.
ggplot(data = se) +
  geom_point(aes(x = year, y = visitors, color = state)) +
  labs(x = "Year",
       y = "Visitation",
       title = "Southeast States National Park Visitation") +
  theme_light() +
  theme(legend.title = element_blank(),
        axis.text.x = element_text(angle = 45, hjust = 1, size = 14))
```



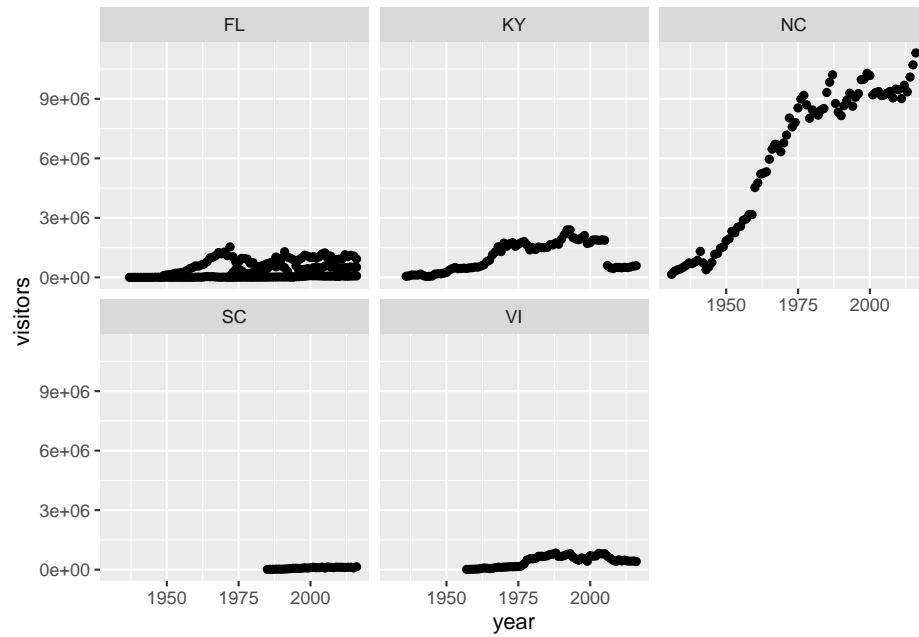
4. The code is missing a geom to describe how the data should be plotted.

STOP: commit, pull and push to github

7.9 Faceting

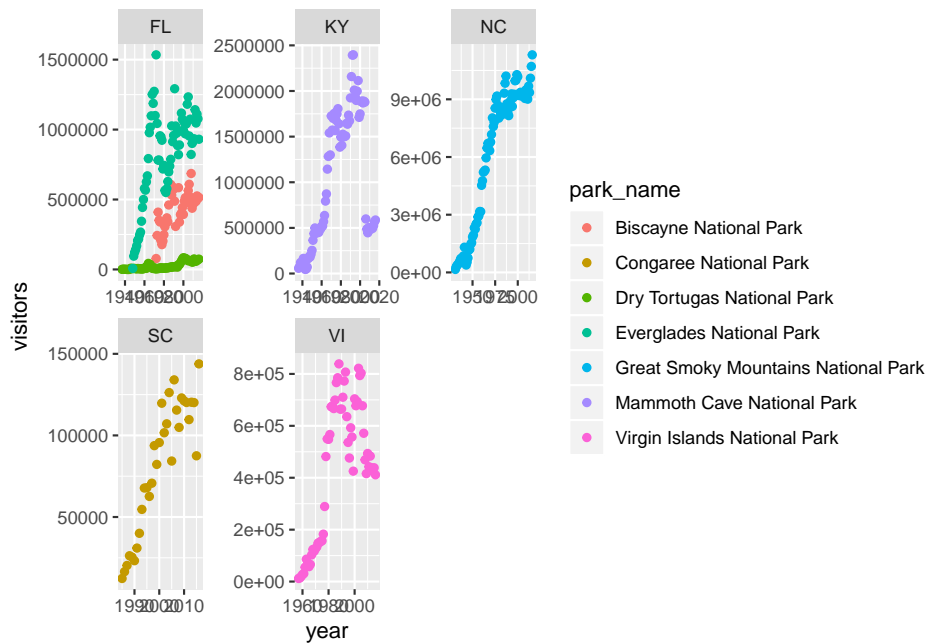
ggplot has a special technique called *faceting* that allows the user to split one plot into multiple plots based on data in the dataset. We will use it to make a plot of park visitation by state:

```
ggplot(data = se) +
  geom_point(aes(x = year, y = visitors)) +
  facet_wrap(~ state)
```



We can now make the faceted plot by splitting further by park using `park_name` (within a single plot):

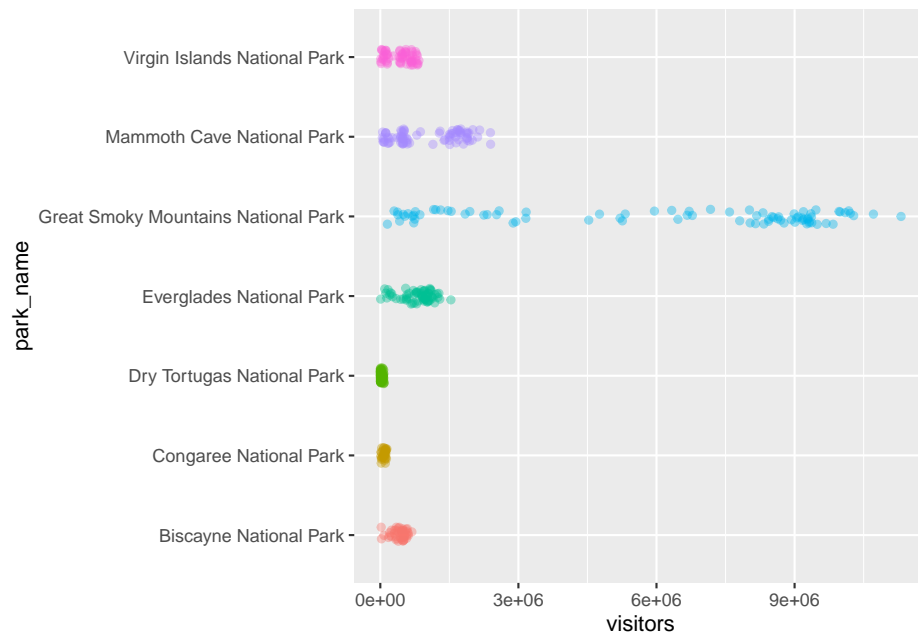
```
ggplot(data = se) +
  geom_point(aes(x = year, y = visitors, color = park_name)) +
  facet_wrap(~ state, scales = "free")
```



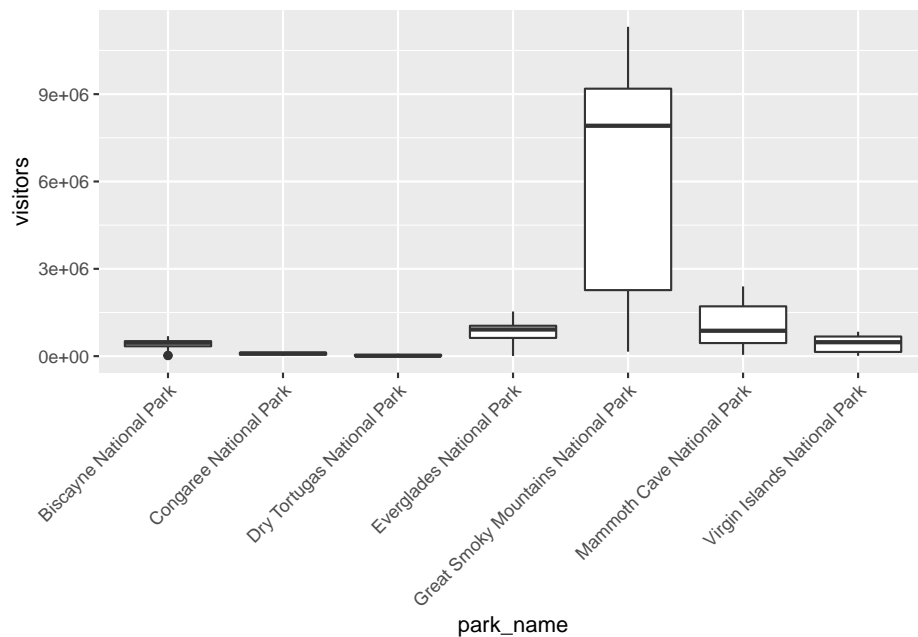
7.10 Geometric objects (geoms)

A **geom** is the geometrical object that a plot uses to represent data. People often describe plots by the type of geom that the plot uses. For example, bar charts use bar geoms, line charts use line geoms, boxplots use boxplot geoms, and so on. Scatterplots break the trend; they use the point geom. You can use different geoms to plot the same data. To change the geom in your plot, change the geom function that you add to `ggplot()`. Let's look at a few ways of viewing the distribution of annual visitation (`visitors`) for each park (`park_name`).

```
ggplot(data = se) +
  geom_jitter(aes(x = park_name, y = visitors, color = park_name),
              width = 0.1,
              alpha = 0.4) +
  coord_flip() +
  theme(legend.position = "none")
```



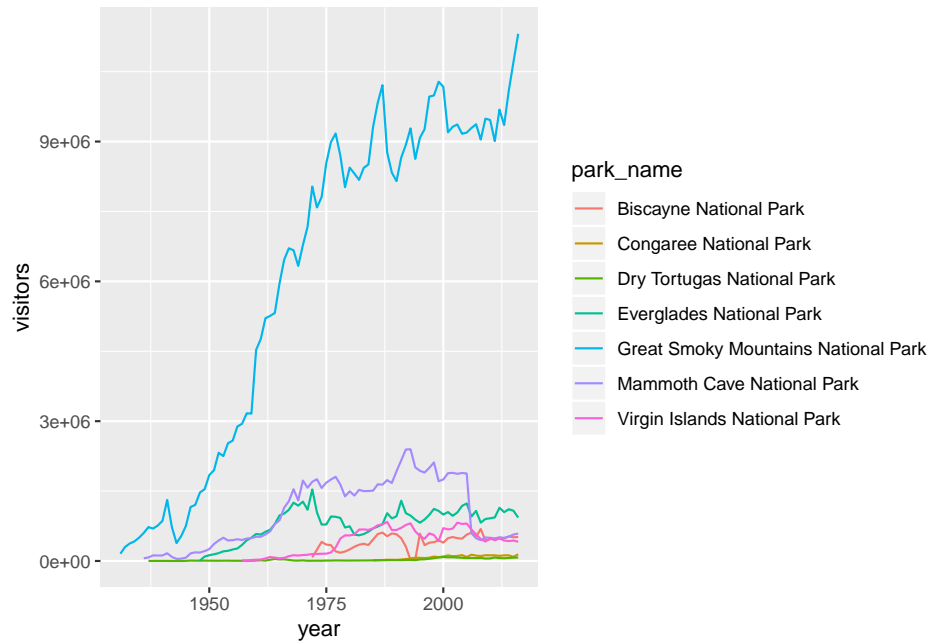
```
ggplot(se, aes(x = park_name, y = visitors)) +
  geom_boxplot() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



None of these are great for visualizing data over time. We can use `geom_line()`

in the same way we used `geom_point`.

```
ggplot(se, aes(x = year, y = visitors, color = park_name)) +  
  geom_line()
```



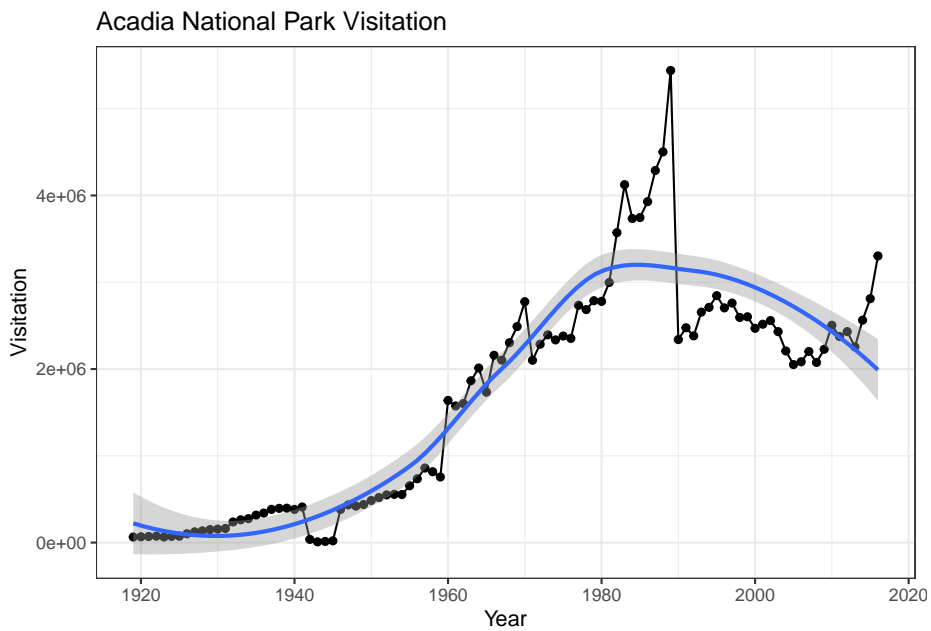
`ggplot2` provides over 30 geoms, and extension packages provide even more (see <https://www.ggplot2-exts.org> for a sampling). The best way to get a comprehensive overview is the `ggplot2` cheatsheet. To learn more about any single geom, use help: `?geom_smooth`.

To display multiple geoms in the same plot, add multiple geom functions to `ggplot()`:

`geom_smooth` allows you to view a smoothed mean of data. Here we look at the smooth mean of visitation over time to Acadia National Park:

```
ggplot(data = acadia) +  
  geom_point(aes(x = year, y = visitors)) +  
  geom_line(aes(x = year, y = visitors)) +  
  geom_smooth(aes(x = year, y = visitors)) +  
  labs(title = "Acadia National Park Visitation",  
        y = "Visitation",  
        x = "Year") +  
  theme_bw()
```

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

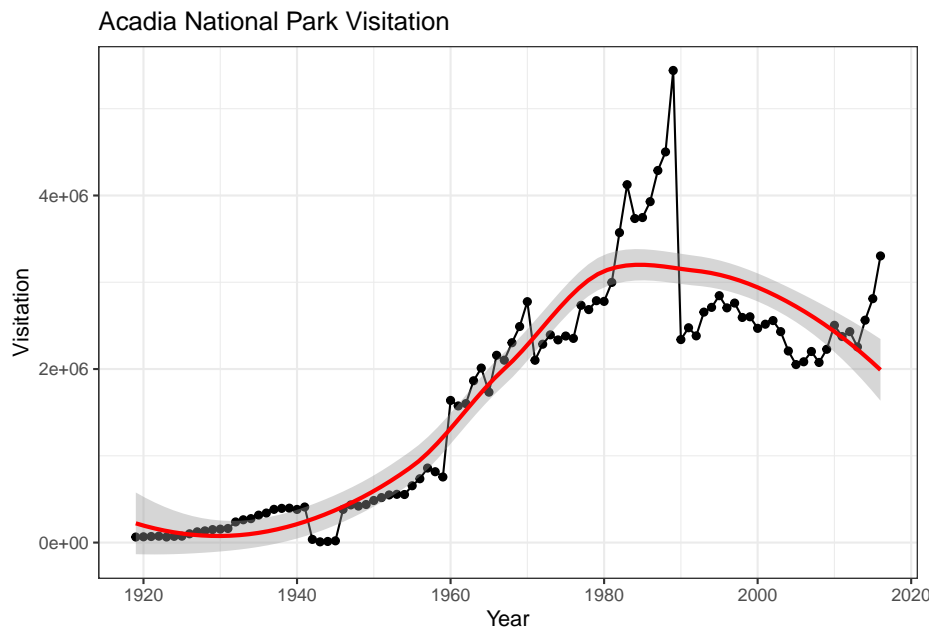


Notice that this plot contains three geoms in the same graph! Each geom is using the set of mappings in the first line. `ggplot2` will treat these mappings as global mappings that apply to each geom in the graph.

If you place mappings in a geom function, `ggplot2` will treat them as local mappings for the layer. It will use these mappings to extend or overwrite the global mappings *for that layer only*. This makes it possible to display different aesthetics in different layers.

```
ggplot(data = acadia, aes(x = year, y = visitors)) +
  geom_point() +
  geom_line() +
  geom_smooth(color = "red") +
  labs(title = "Acadia National Park Visitation",
        y = "Visitation",
        x = "Year") +
  theme_bw()
```

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



7.10.1 Challenge

With all of this information in hand, please take another five minutes to either improve one of the plots generated in this exercise or create a beautiful graph of your own. Use the RStudio **ggplot2** cheat sheet for inspiration.

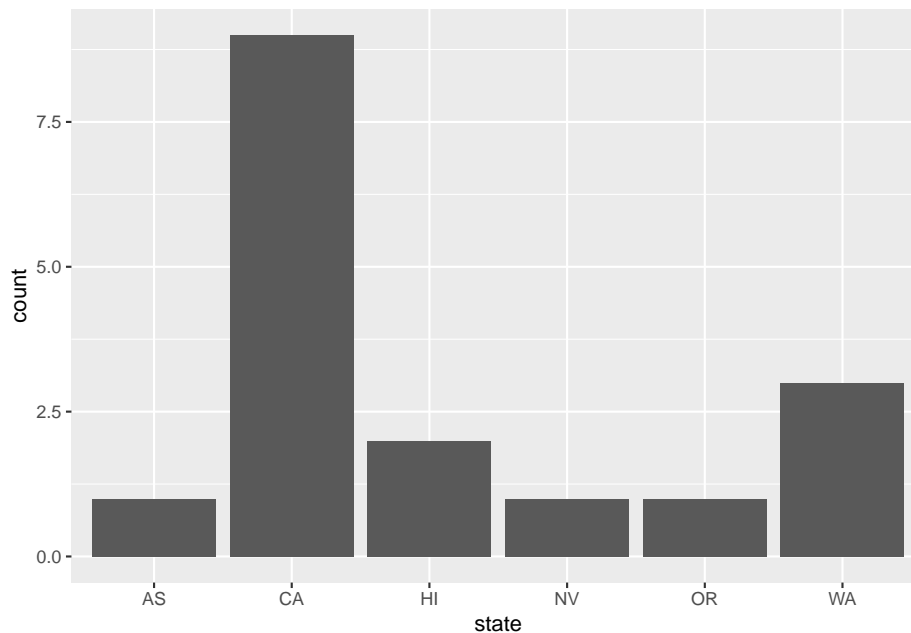
Here are some ideas:

- See if you can change the thickness of the lines or line type (e.g. dashed line).
- Can you find a way to change the name of the legend? What about its labels?
- Try using a different color palette (see [http://www.cookbook-r.com/Graphs/Colors_\(ggplot2\)/](http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/)).

7.11 Bar charts

Next, let's take a look at a bar chart. Bar charts seem simple, but they are interesting because they reveal something subtle about plots. Consider a basic bar chart, as drawn with `geom_bar()`. The following chart displays the total number of parks in each state within the Pacific West region.

```
ggplot(data = visit_16, aes(x = state)) +  
  geom_bar()
```



On the x-axis, the chart displays **state**, a variable from **visit_16**. On the y-axis, it displays **count**, but **count** is not a variable in **visit_16**! Where does **count** come from? Many graphs, like scatterplots, plot the raw values of your dataset. Other graphs, like bar charts, calculate new values to plot:

- bar charts, histograms, and frequency polygons bin your data and then plot bin counts, the number of points that fall in each bin.
- smoothers fit a model to your data and then plot predictions from the model.
- boxplots compute a robust summary of the distribution and then display a specially formatted box.

The algorithm used to calculate new values for a graph is called a **stat**, short for statistical transformation.

You can learn which stat a geom uses by inspecting the default value for the **stat** argument. For example, `?geom_bar` shows that the default value for **stat** is “count”, which means that `geom_bar()` uses `stat_count()`. `stat_count()` is documented on the same page as `geom_bar()`, and if you scroll down you can find a section called “Computed variables”. That describes how it computes two new variables: **count** and **prop**.

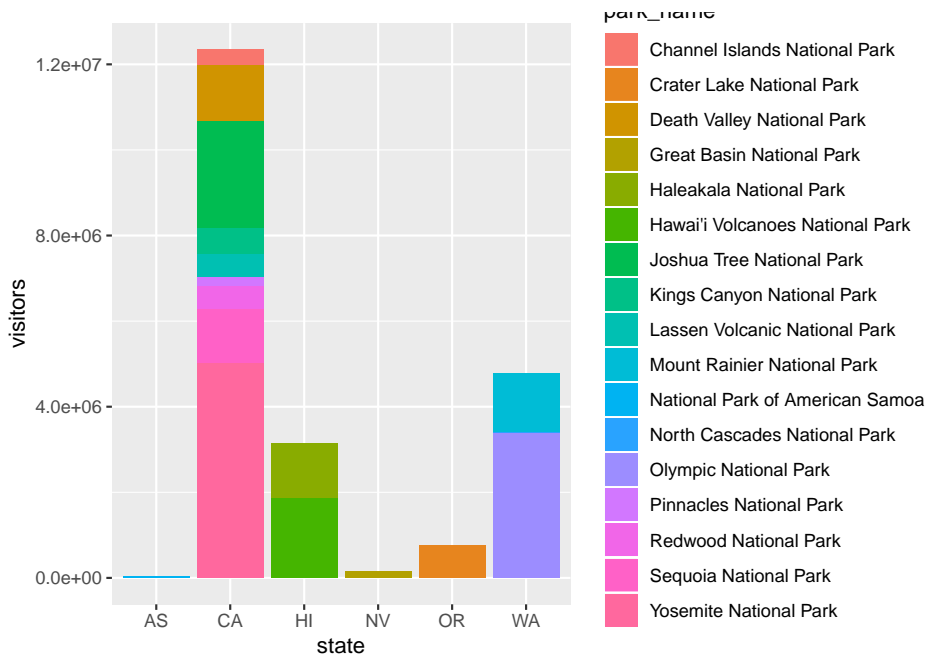
ggplot2 provides over 20 stats for you to use. Each stat is a function, so you

can get help in the usual way, e.g. `?stat_bin`. To see a complete list of stats, try the `ggplot2` cheatsheet.

7.11.1 Position adjustments

There's one more piece of magic associated with bar charts. You can colour a bar chart using either the `color` aesthetic, or, more usefully, `fill`:

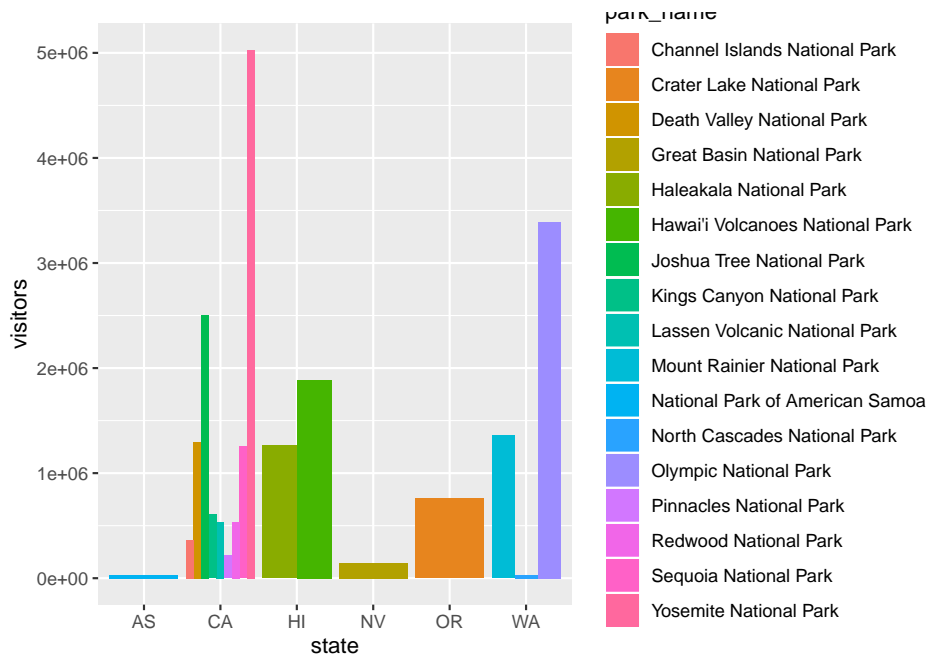
```
ggplot(data = visit_16, aes(x = state, y = visitors, fill = park_name)) +  
  geom_bar(stat = "identity")
```



The stacking is performed automatically by the **position adjustment** specified by the `position` argument. If you don't want a stacked bar chart, you can use `"dodge"`.

- `position = "dodge"` places overlapping objects directly *beside* one another. This makes it easier to compare individual values.

```
ggplot(data = visit_16, aes(x = state, y = visitors, fill = park_name)) +  
  geom_bar(stat = "identity", position = "dodge")
```



7.11.2 Challenge

With all of this information in hand, please take another five minutes to either improve one of the plots generated in this exercise or create a beautiful graph of your own. Use the RStudio **ggplot2** cheat sheet for inspiration. Remember to use the help documentation (e.g. `?geom_bar`)

Here are some ideas:

- Flip the x and y axes.
- Change the color palette used
- Use `scale_x_discrete` to change the x-axis tick labels to the full state names (Arizona, Colorado, etc.)
- Make a bar chart using the Massachusetts dataset (`mass`) and find out how many parks of each type are in the state.

7.11.2.0.1 Answers (no peeking!)

```
#how many of each types of parks are in Massachusetts?
ggplot(data = mass) +
  geom_bar(aes(x = type, fill = park_name)) +
  labs(x = "",
```



7.12 Arranging and exporting plots

After creating your plot, you can save it to a file in your favorite format. The Export tab in the **Plot** pane in RStudio will save your plots at low resolution, which will not be accepted by many journals and will not scale well for posters.

Instead, use the `ggsave()` function, which allows you easily change the dimension and resolution of your plot by adjusting the appropriate arguments (`width`, `height` and `dpi`):

```

my_plot <- ggplot(data = mass) +
  geom_bar(aes(x = type, fill = park_name)) +
  labs(x = "",
       y = "")+
  theme(axis.text.x = element_text(angle = 45, hjust = 1, size = 7))

ggsave("name_of_file.png", my_plot, width = 15, height = 10)

```

Note: The parameters `width` and `height` also determine the font size in the saved plot.

7.13 Bonus

So as you can see, `ggplot2` is a fantastic package for visualizing data. But there are some additional packages that let you make plots interactive. `plotly`, `gganimate`.

```
#install.packages("plotly")
library(plotly)

ggplotly(my_plot)

acad_vis <- ggplot(data = acadia, aes(x = year, y = visitors)) +
  geom_point() +
  geom_line() +
  geom_smooth(color = "red") +
  labs(title = "Acadia National Park Visitation",
       y = "Visitation",
       x = "Year") +
  theme_bw()

ggplotly(acad_vis)
```

7.14 Save and push to GitHub

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

7.16 Interludes (deep thoughts/openscapes)

7.17 Our Turn Your Turn 1

7.18 Our Turn Your Turn 2

7.19 Efficiency Tips

Chapter 8

dplyr and Pivot Tables

TODO: use lobster data. Start with counted data and show uncount and discuss why it's important to have counted data Count and uncount, summary statistics Use uncounted lobster data summarize(mean, sd, n) with uncounted lobster data (stdev in excel?) DT, kable — introduce this and then will be repeated in tidying if time, could we show count vs n but if you want to get other statistics

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)

8.1 Summary (a few sentences)

We will learn how to wrangle data in R, using the `dplyr` package which is included in the `tidyverse` package. In this session, we'll focus on the functions in `dplyr` that operate like Excel's pivot tables.

8.2 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.

- Practice our reproducible workflow with RMarkdown and GitHub
- Discuss pivot tables in Excel
- Introduce the `dplyr` package in R

8.3 Resources

- dplyr.tidyverse.org
- R for Data Science: Transform Chapter
- Intro to Pivot Tables I-III by Excel Campus

8.4 Lesson

TODO We've talked about

8.4.1 Setup

Let's start a new RMarkdown file, called X.

In the setup chunk, let's attach our libraries and read in the lobster counts data.

```
## attach libraries
library(tidyverse)

## read in data
lobster_counts <- read_csv("lobster_counts_curated.csv")
```

Let's add a code chunk to explore the data by looking at some summary statistics and making a simple plot.

```
#TODO
head(lobster_counts) # year and month as well as a column for date
```

```
## # A tibble: 6 x 10
##   year month date   site transect replicate size_mm count num_ao area
##   <dbl> <dbl> <chr>  <chr>   <dbl> <chr>         <dbl> <dbl>  <dbl> <dbl>
## 1  2012     8 8/20/12 ivee      1 A             NA     0     0    300
## 2  2012     8 8/20/12 ivee      1 B             NA     0     0    300
## 3  2012     8 8/20/12 ivee      1 C             NA     0     0    300
## 4  2012     8 8/20/12 ivee      1 D             NA     0     0    300
## 5  2012     8 8/20/12 ivee      2 A             NA     0     0    300
## 6  2012     8 8/20/12 ivee      2 B             NA     0     0    300
```

```
summary(lobster_counts)
```

```
##           year           month           date           site
##  Min.   :2012   Min.   :8.000   Length:4362   Length:4362
##  1st Qu.:2015   1st Qu.:8.000   Class :character   Class :character
##  Median :2016   Median :8.000   Mode  :character   Mode  :character
##  Mean    :2016   Mean    :8.018
##  3rd Qu.:2018   3rd Qu.:8.000
```

```
## Max. :2018 Max. :9.000
##
##      transect      replicate      size_mm      count
## Min. :1.000 Length:4362 Min. : 18.00 Min. : 0.000
## 1st Qu.:2.000 Class :character 1st Qu.: 65.00 1st Qu.: 1.000
## Median :4.000 Mode :character Median : 75.00 Median : 1.000
## Mean :4.056 Mean : 74.45 Mean : 1.459
## 3rd Qu.:6.000 3rd Qu.: 84.00 3rd Qu.: 2.000
## Max. :9.000 Max. :183.00 Max. :41.000
## NA's :350
##      num_ao      area
## Min. : 0.00000 Min. :300
## 1st Qu.: 0.00000 1st Qu.:300
## Median : 0.00000 Median :300
## Mean : 0.04104 Mean :300
## 3rd Qu.: 0.00000 3rd Qu.:300
## Max. :10.00000 Max. :300
##
# summary, head, tail, depending on readxl
# ggplot
```

Also explore in the Viewer

OK now let's step back from R and discuss pivot tables in Excel.

8.4.2 Pivot tables

8.4.2.1 What are they?

TODO: screenshots, demo with lobster data.

What is actually going on is that they are summarizing by the groups you identify.

So I'm looking at my lobster data in Excel and I really want to know how many lobsters were counted at each site. I want a summary of total counts by site. So to do this in Excel we would initiate the Pivot Table Process:

And it will do its best to find the data I would like to include in my Pivot Table (it can have difficulty with non-rectangular or “non-tidy” data), and suggest we make this in a new sheet:

And then we'll get a little wizard to help us create the Pivot Table. I want to summarize by site, so I drag “site” down into the “Rows” box, and then I drag “count” into the “Values” box. And it will create a Pivot Table for me, with “sum” as the default summary statistic.

A few things to note:

- The pivot table is separate entity from our data (it's on a different sheet); the original data has not been affected
- The pivot table only shows the variables we requested; we don't see other columns (like date, month, or site).

8.4.2.2 Why are they great?

Pivot tables are great because they summarize the data and keep the raw data raw — they even promote good practice because they by default ask you if you'd like to present the data in a new sheet rather than in the same sheet.

If you add new data, you can refresh your table

8.4.2.3 Why would we want to work in R instead?

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

8.5 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
```

8.6 `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.

8.6.1 `group_by` one variable

Let’s try this on our `lobster_counts` data. Let’s calculate the the total number of lobster by year. In R-speak, we will `group_by` year and then `summarize` by calculating the sum of count. We’ll use the pipe operator `%>%`

```
lobster_counts %>%
  group_by(year) %>%
  summarize(total_lobster = sum(count)) # within summarize, we name a new column and calculate the
```

```
## # A tibble: 7 x 2
##   year total_lobster
##   <dbl>         <dbl>
## 1  2012             231
## 2  2013             243
## 3  2014             510
## 4  2015            1100
## 5  2016             809
## 6  2017            1668
## 7  2018            1805
```

This returns output summarizing the `total_lobster` for each year, just like we saw in the pivot table in Excel.

Notice how together, `group_by` and `summarize` minimize the amount of information we see. We also saw this with the pivot table. We lose the other columns

that aren't involved here.

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

```
lobster_counts %>%
  summarize(total_lobster = sum(count))
```

```
## # A tibble: 1 x 1
##   total_lobster
##           <dbl>
## 1           6366
```

So if we don't `group_by` first, we will get a single summary statistic (sum 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. (This is the same behavior you would get in a pivot table if you removed `year` from the "Rows" field, leaving only `count` in the "Values" field).

Let's now check the `lobster_counts` variable. We can do this by clicking on `lobster_counts` in the Environment pane in RStudio.

We see that we haven't changed any of our original data that was stored in this variable. (Just like how the pivot table didn't affect the raw data on the original sheet).

Aside: You'll also see that when you click on the variable name in the Environment pane, `View(lobster_counts)` shows up in your Console. `View()` (capital V) is the R function to view any variable in the viewer. So this is something that you can write in your RMarkdown script, although RMarkdown will not be able to knit this view feature into the formatted document. So, if you want include `View()` in your RMarkdown document you will need to either comment it out `#View()` or add `eval=FALSE` to the top of the code chunk so that the full line reads `{r, eval=FALSE}`.

So we can make the equivalent of Excel's pivot table in R with `group_by` and then `summarize`. But a powerful thing about R is that maybe we want this information to be used in further analyses. We can make this easier for ourselves by saving this as a variable. So let's add a variable assignment to that first line:

```
year_summary <- lobster_counts %>%
  group_by(year) %>%
  summarize(total_lobster = sum(count))
```

8.6.2 Activity

Summarize `lobster_counts` by site and assign it to a variable called `site_summary`.

```
site_summary <- lobster_counts %>%
  group_by(site) %>%
  summarize(total_lobster = sum(count))
```

8.6.3 group_by multiple variables

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 <- lobster_counts %>%
  group_by(site, year) %>%
  summarize(total_lobster = sum(count))
```

```
site_year_summary
```

```
## # A tibble: 35 x 3
## # Groups:   site [5]
##   site  year total_lobster
##   <chr> <dbl>         <dbl>
## 1 aque  2012             38
## 2 aque  2013             32
## 3 aque  2014            100
## 4 aque  2015             83
## 5 aque  2016             48
## 6 aque  2017             67
## 7 aque  2018             54
## 8 carp  2012             78
## 9 carp  2013             93
## 10 carp 2014             79
## # ... with 25 more rows
```

Let's do this quickly in Excel for comparison. We can drag site to our previous pivot table:

Notice that in Excel we retain the overall totals for each site (in bold, on the same line with the site name). This might be nice at a glance right now, but it can be problematic. Why? What if someone unfamiliar with pivot tables sums this whole column? There are not a lot of safeguards here.

So getting back to R, 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.

8.7 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 because it's not a numeric value:

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

```
## # A tibble: 4,362 x 11
##   year month date site transect replicate size_mm count num_ao area
##   <dbl> <dbl> <chr> <chr>   <dbl> <chr>      <dbl> <dbl> <dbl> <dbl>
## 1  2012     8 8/20~ ivee      1 A         NA      0      0    300
## 2  2012     8 8/20~ ivee      1 B         NA      0      0    300
## 3  2012     8 8/20~ ivee      1 C         NA      0      0    300
## 4  2012     8 8/20~ ivee      1 D         NA      0      0    300
## 5  2012     8 8/20~ ivee      2 A         NA      0      0    300
## 6  2012     8 8/20~ ivee      2 B         NA      0      0    300
## 7  2012     8 8/20~ ivee      2 C         NA      0      0    300
## 8  2012     8 8/20~ ivee      2 D         NA      0      0    300
## 9  2012     8 8/20~ ivee      3 A         70      1      0    300
## 10 2012     8 8/20~ ivee      3 B         60      1      0    300
## # ... with 4,352 more rows, and 1 more variable: observation_type <chr>
```

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_lobster`:

```
lobster_counts %>%
  mutate(total_lobster = sum(count))
```

```
## # A tibble: 4,362 x 11
##   year month date site transect replicate size_mm count num_ao area
##   <dbl> <dbl> <chr> <chr>   <dbl> <chr>      <dbl> <dbl> <dbl> <dbl>
## 1  2012     8 8/20~ ivee      1 A         NA      0      0    300
## 2  2012     8 8/20~ ivee      1 B         NA      0      0    300
```

```
## 3 2012      8 8/20~ ivee          1 C          NA      0      0    300
## 4 2012      8 8/20~ ivee          1 D          NA      0      0    300
## 5 2012      8 8/20~ ivee          2 A          NA      0      0    300
## 6 2012      8 8/20~ ivee          2 B          NA      0      0    300
## 7 2012      8 8/20~ ivee          2 C          NA      0      0    300
## 8 2012      8 8/20~ ivee          2 D          NA      0      0    300
## 9 2012      8 8/20~ ivee          3 A          70      1      0    300
## 10 2012     8 8/20~ ivee          3 B          60      1      0    300
## # ... with 4,352 more rows, and 1 more variable: total_lobster <dbl>
```

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.

8.7.1 Activity

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

8.7.2 `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.

```
lobster_counts %>%
  group_by(site, year) %>%
  mutate(siteyear_counts = sum(count))
```

```
## # A tibble: 4,362 x 11
## # Groups:   site, year [35]
##   year month date site transect replicate size_mm count num_ao area
##   <dbl> <dbl> <chr> <chr>   <dbl> <chr>      <dbl> <dbl> <dbl> <dbl>
## 1 2012      8 8/20~ ivee         1 A          NA      0      0    300
## 2 2012      8 8/20~ ivee         1 B          NA      0      0    300
## 3 2012      8 8/20~ ivee         1 C          NA      0      0    300
## 4 2012      8 8/20~ ivee         1 D          NA      0      0    300
## 5 2012      8 8/20~ ivee         2 A          NA      0      0    300
## 6 2012      8 8/20~ ivee         2 B          NA      0      0    300
## 7 2012      8 8/20~ ivee         2 C          NA      0      0    300
## 8 2012      8 8/20~ ivee         2 D          NA      0      0    300
## 9 2012      8 8/20~ ivee         3 A          70      1      0    300
## 10 2012     8 8/20~ ivee         3 B          60      1      0    300
```

```
## # ... with 4,352 more rows, and 1 more variable: siteyear_counts <dbl>
```

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.

8.8 `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 lobster 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()`.

```
lobster_percs <- lobster_counts %>%
  group_by(site, year) %>%
  mutate(siteyear_counts = sum(count)) %>%
  mutate(perc_lobster = 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.

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 lobster siteyear_counts
x <- lobster_counts %>%
  group_by(site, year) %>%
  summarize(siteyear_counts = sum(count))

## then somehow join or merge that information to the lobster_counts data
x %>%
  mutate(perc_lobster = 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 `lobster_counts`. This actually requires a few more dplyr verbs: `filter` and `*_join`; we will do this tomorrow!

count column (would have to uncount first)

8.9 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.”

8.10 Efficiency Tips

arrow keys with shift, option, command

Chapter 9

Tidying

9.1 Summary

In previous sessions, we learned to read in data, do some wrangling, and create a graph and table. Here, we'll continue by *reshaping* data frames (converting from long-to-wide, or wide-to-long format), *separating* and *uniting* variable (column) contents, converting between *explicit* and *implicit* missing (NA) values, and cleaning up our column names with the `janitor` package.

9.2 Objectives

- Reshape data frames with `tidyr::pivot_wider()` and `tidyr::pivot_longer()`
- Convert column names with `janitor::clean_names()`
- Combine or separate information from columns with `tidyr::unite()` and `tidyr::separate()`
- Make implicit missings *explicit* with `tidyr::complete()`
- Make explicit missings *implicit* with `tidyr::drop_na()`
- Use our new skills as part of a bigger wrangling sequence
- Make a customized table (TODO: or introduce Kable if not time in pivot tables chapter)

9.3 Resources

– Ch. 12 *Tidy Data*, in R for Data Science by Grolemund & Wickham - `tidyr` documentation from tidyverse.org - `janitor` repo / information from Sam Firke

9.4 Lesson

9.4.1 Lesson Prep

9.4.1.1 Create a new R Markdown and attach packages

Within your day 2 R Project, create a new .Rmd. Attach the `tidyverse`, `janitor` and `readxl` packages with `library(package_name)`. Knit and save your new .Rmd within the project folder.

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

9.4.1.2 Read in data

Use `readxl::read_excel()` to import the “invert_counts_curated.xlsx” data:

```
inverts_df <- readxl::read_excel("invert_counts_curated.xlsx")
```

Be sure to explore the imported data a bit:

- `View()`
- `names()`
- `summary()`

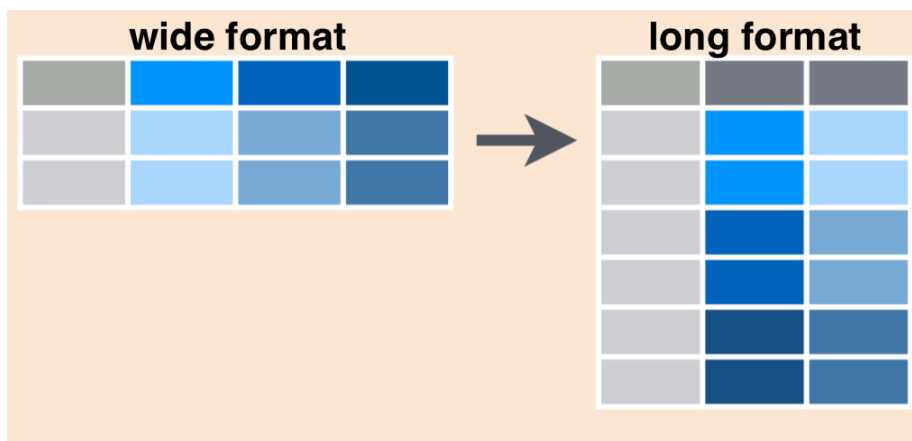
9.4.2 Reshaping with `tidyr::pivot_longer()` and `tidyr::pivot_wider()`

9.4.2.1 Wide-to-longer format with `tidyr::pivot_longer()`

In *tidy format*, each variable is contained within a single column. If we look at `inverts_df`, we can see that the *year* variable is actually split over 3 columns, so we’d say this is currently in **wide format**.

There may be times when you want to have data in wide format, but often with code it is more efficient to convert to **long format** by gathering together observations for a variable that is currently split into multiple columns.

Schematically, converting from wide to long format looks like this:



Generally, the code to gather wide columns together using `tidyr::pivot_longer()` looks like this:

TODO: Add `pivot_longer()` schematic

We'll use `tidyr::pivot_longer()` to gather data from all years in *inverts_df* into two columns: one called *year*, which contains the year (as a number), and another called *sp_count* that contains the number of each species observed. The new data frame will be stored as *inverts_long*:

```
inverts_long <- tidyr::pivot_longer(data = invert_df,
                                   cols = '2016':'2018',
                                   names_to = "year",
                                   values_to = "sp_count")
```

The outcome is the new long-format *inverts_long* data frame:

```
inverts_long
```

```
## # A tibble: 165 x 5
##   month site common_name      year sp_count
##   <chr> <chr> <chr>         <chr>   <dbl>
## 1 7     abur  california cone snail  2016     451
## 2 7     abur  california cone snail  2017      28
## 3 7     abur  california cone snail  2018    762
## 4 7     abur  california spiny lobster 2016      17
## 5 7     abur  california spiny lobster 2017      17
## 6 7     abur  california spiny lobster 2018      16
## 7 7     abur  orange cup coral    2016      24
## 8 7     abur  orange cup coral    2017      24
## 9 7     abur  orange cup coral    2018      24
## 10 7    abur  purple urchin      2016      48
## # ... with 155 more rows
```

Hooray, long format!

One thing that isn't obvious at first (but would become obvious if you continued working with this data) is that since those year numbers were initially column names (characters), when they are stacked into the *year* column, their class wasn't auto-updated to numeric.

Explore the class of *year* in *inverts_long*:

```
class(inverts_long$year)
```

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

We'll use `dplyr::mutate()` in a different way here: to create a new column (that's how we've used `mutate()` previously) that has the same name of an existing column, in order to update and overwrite the existing column.

In this case, we'll `mutate()` to add a column called *year*, which contains an `as.numeric()` version of the existing *year* variable:

```
# Coerce "year" class to numeric:
```

```
inverts_long <- invert_long %>%  
  mutate(year = as.numeric(year))
```

Checking the class again, we see that *year* has been updated to a numeric variable:

```
class(inverts_long$year)
```

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

9.4.2.2 Long-to-wider format with `tidyr::pivot_wider()`

In the previous example, we had information spread over multiple columns that we wanted to *gather*. Sometimes, we'll have data that we want to *spread* over multiple columns.

For example, imagine that starting from *inverts_long* we want each species in the *common_name* column to exist as its **own column**. In that case, we would be converting from a longer to a wider format, and will use `tidyr::pivot_wider()` as follows:

TODO: Add `pivot_wider()` schematic

Specifically for our data, we write code to spread the *common_name* column as follows:

```
inverts_wide <- inverts_long %>%  
  tidyr::pivot_wider(names_from = common_name,  
                     values_from = sp_count)
```

```
inverts_wide
```

```
## # A tibble: 33 x 8
##   month site   year `california con~` `california spi~` `orange cup cor~`
##   <chr> <chr> <dbl>         <dbl>         <dbl>         <dbl>
## 1 7      abur   2016          451             17             24
## 2 7      abur   2017           28             17             24
## 3 7      abur   2018          762             16             24
## 4 7      ahnd   2016           27             16             24
## 5 7      ahnd   2017           24             16             24
## 6 7      ahnd   2018           24             16             24
## 7 7      aque   2016         4971             48            1526
## 8 7      aque   2017         1752             48            1623
## 9 7      aque   2018         2616             48            1859
## 10 7     bull   2016         1735             24             36
## # ... with 23 more rows, and 2 more variables: `purple urchin` <dbl>,
## #   `rock scallop` <dbl>
```

We can see that now each *species* has its own column (wider format). But also notice that those column headers (since they have spaces) might not be in the most coder-friendly format...

9.4.2.3 Meet the janitor package

The `janitor` package by Sam Firke is a brilliant collection of functions for some quick data cleaning. We recommend that you explore the different functions it contains. Like:

- `janitor::clean_names()`: update column headers to a case of your choosing
- `janitor::get_dupes()`: see all rows that are duplicates within variables you choose
- `janitor::remove_empty()`: remove empty rows and/or columns
- `janitor::andorn_*`: jazz up frequency tables of counts (we'll return to this for a table example in TODO: Session 8)
- ...and more!

Here, we'll use `janitor::clean_names()` to convert all of our column headers to a more convenient case - the default is **lower_snake_case**, which means all spaces and symbols are replaced with an underscore (or a word describing the symbol), all characters are lowercase, and a few other nice adjustments.

For example, `janitor::clean_names()` would update these nightmare column names into much nicer forms:

- `My...RECENT-income!` becomes `my_recent_income`
- `SAMPLE2.!test1` becomes `sample2_test1`

- ThisIsTheName becomes this_is_the_name
- 2015 becomes x2015

If we wanted to then use these columns (which we probably would, since we created them), we could clean the names to get them into more coder-friendly lower_snake_case with `janitor::clean_names()`:

```
inverts_wide <- invert_wide %>%
  janitor::clean_names()
```

```
names(inverts_wide)
```

```
## [1] "month"           "site"
## [3] "year"            "california_cone_snail"
## [5] "california_spiny_lobster" "orange_cup_coral"
## [7] "purple_urchin"    "rock_scallop"
```

And there are other options for the case, like:

- “snake” produces snake_case
- “lower_camel” or “small_camel” produces lowerCamel
- “upper_camel” or “big_camel” produces UpperCamel
- “screaming_snake” or “all_caps” produces ALL_CAPS
- “lower_upper” produces lowerUPPER
- “upper_lower” produces UPPERlower

9.4.3 Combine or separate information in columns with `tidyr::unite()` and `tidyr::separate()`

Sometimes we’ll want to *separate* contents of a single column into multiple columns, or *combine* entries from different columns into a single column.

For example, the following data frame has *genus* and *species* in separate columns:

```
id
genus
species
common_name
1
Scorpaena
guttata
sculpin
2
Sebastes
```

miniatus

vermillion

We may want to combine the genus and species into a single column, *scientific_name*:

id

scientific_name

common_name

1

Scorpaena guttata

sculpin

2

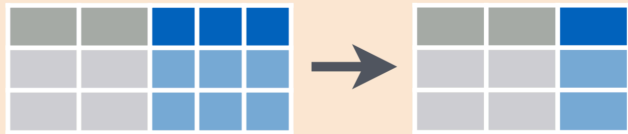
Sebastes miniatus

vermillion

Or we may want to do the reverse (separate information from a single column into multiple columns). Here, we'll learn `tidyr::unite()` and `tidyr::separate()` to help us do both.

9.4.3.1 `tidyr::unite()` to merge information from separate columns

Use `tidyr::unite()` to combine (paste) information from multiple columns into a single column (as for the scientific name example above)



`tidyr::unite(data, col, ..., sep)`

Unite several columns into one.

To demonstrate uniting information from separate columns, we'll make a single column that has the combined information from *site* abbreviation and *year* in *inverts_wide*.

We need to give `tidyr::unite()` several arguments:

- **data:** the data frame containing columns we want to combine (or pipe into the function from the data frame)
- **col:** the name of the new “united” column
- the **columns you are uniting**
- **sep:** the symbol, value or character to put between the united information from each column

```
inverts_unite <- inverts_wide %>%
  tidyr::unite(col = "site_year", # What to name the new united column
              c(site, year), # The columns we'll unite (site, year)
              sep = "_") # How to separate the things we're uniting
```

```
## # A tibble: 6 x 7
##   month site_year california_cone~ california_spin~ orange_cup_coral
##   <chr> <chr>          <dbl>          <dbl>          <dbl>
## 1 7      abur_2016          451            17            24
## 2 7      abur_2017           28            17            24
## 3 7      abur_2018          762            16            24
## 4 7      ahnd_2016           27            16            24
## 5 7      ahnd_2017           24            16            24
## 6 7      ahnd_2018           24            16            24
## # ... with 2 more variables: purple_urchin <dbl>, rock_scallop <dbl>
```

Try updating the separator from “_” to “hello!” to see what the outcome column contains.

`tidyr::unite()` can also combine information from *more* than two columns. For example, to combine the *site*, *common_name* and *year* columns from *inverts_long*, we could use:

```
# Uniting more than 2 columns:

inverts_triple_unite <- inverts_long %>%
  tidyr::unite(col = "year_site_name",
              c(year, site, common_name),
              sep = "-")
```

```
head(inverts_triple_unite)
```


```
## # A tibble: 6 x 3
##   month year_site_name          sp_count
##   <chr> <chr>          <dbl>
## 1 7      2016-abur-california cone snail      451
## 2 7      2017-abur-california cone snail       28
## 3 7      2018-abur-california cone snail      762
## 4 7      2016-abur-california spiny lobster    17
## 5 7      2017-abur-california spiny lobster    17
## 6 7      2018-abur-california spiny lobster    16
```


9.4.3.2 `tidyr::separate()` to separate information into multiple columns

While `tidyr::unite()` allows us to combine information from multiple columns, it's more likely that you'll *start* with a single column that you want to split up into pieces.

For example, I might want to split up a column containing the *genus* and *species* (*Scorpaena guttata*) into two separate columns (*Scorpaena* | *guttata*), so that I can count how many *Scorpaena* organisms exist in my dataset at the genus level.

Use `tidyr::separate()` to “separate a character column into multiple columns using a regular expression separator.”



tidyr::separate(storms, date, c("y", "m", "d"))
Separate one column into several.

Let's start again with *inverts_unite*, where we have combined the *site* and *year* into a single column called *site_year*. If we want to **separate** those, we can use:

```
inverts_sep <- invert_triplet_unite %>%
  tidyr::separate(year_site_name, into = c("my_year", "my_site_name"))
```

```
## Warning: Expected 2 pieces. Additional pieces discarded in 165 rows [1, 2,
## 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].
```

What is that warning *Expected 2 pieces...* telling us? If we take a look at the resulting data frame *inverts_sep*, we see that it only keeps the first **two** pieces, and gets rid of the third (name). Which is a bit concerning, because we rarely want to just throw away information in a data frame.

```
head(inverts_sep)
```

```
## # A tibble: 6 x 4
##   month my_year my_site_name sp_count
##   <chr> <chr>    <chr>         <dbl>
## 1 7      2016     abur           451
## 2 7      2017     abur            28
## 3 7      2018     abur          762
## 4 7      2016     abur            17
## 5 7      2017     abur            17
## 6 7      2018     abur            16
```

That's problematic. How can we make sure we're keeping as many different elements as exist in the united column?

We have a couple of options:

1. Create the *number* of columns that are needed to retain as many elements as exist (in this case, 3, but we only created two new columns in the example above)

```
inverts_sep3 <- inverters_triple_unite %>%
  tidyr::separate(year_site_name, into = c("the_year", "the_site", "the_name"))
```

```
## Warning: Expected 3 pieces. Additional pieces discarded in 165 rows [1, 2,
## 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].
```

Another warning. What is that about? Let's take a look at the resulting data frame and think about what's missing (what are the "pieces discarded"?):

```
head(inverts_sep3)
```

```
## # A tibble: 6 x 5
##   month the_year the_site the_name  sp_count
##   <chr> <chr>    <chr>   <chr>    <dbl>
## 1 7      2016     abur    california  451
## 2 7      2017     abur    california   28
## 3 7      2018     abur    california  762
## 4 7      2016     abur    california   17
## 5 7      2017     abur    california   17
## 6 7      2018     abur    california   16
```

Aha! Only the *first word* of the common name was retained, and anything else was trashed. We want to keep everything after the second dash in the new *the_name* column.

That's because the **default is extra = "warn"**, which means that if you have more pieces than columns you're separating into, it will populate the columns that have been allotted (in our case, just 3) then drop any additional information, giving you a warning that pieces have been dropped.

To keep the extra pieces that have been dropped, add the **extra = "merge"** argument within `tidyr::separate()` to override:

```
inverts_sep_all <- inverters_triple_unite %>%
  separate(year_site_name,
    into = c("sample_year", "location", "sp_name"),
    extra = "merge")
```

No warning there about things being discarded. Explore *inverts_sep_all*:

```
## # A tibble: 165 x 5
##   month sample_year location sp_name          sp_count
```

```
##      <chr> <chr>      <chr>      <chr>      <dbl>
##    1 7      2016      abur      california cone snail      451
##    2 7      2017      abur      california cone snail      28
##    3 7      2018      abur      california cone snail      762
##    4 7      2016      abur      california spiny lobster      17
##    5 7      2017      abur      california spiny lobster      17
##    6 7      2018      abur      california spiny lobster      16
##    7 7      2016      abur      orange cup coral      24
##    8 7      2017      abur      orange cup coral      24
##    9 7      2018      abur      orange cup coral      24
##   10 7      2016      abur      purple urchin      48
## # ... with 155 more rows
```

We see that the resulting data frame has split *year_site_name* into three separate columns, *sample_year*, *location*, and *sp_name*, but now everything after the second break (“-”) remains together in *sp_name* instead of dropping pieces following the third word.

9.4.4 Convert between explicit and implicit missings (NAs)

An *explicit missing* is when every possible outcome actually appears in a data frame as a row, even if a variable of interest for that row is missing (NA).

Conversely, an *implicit missing* is when an observation (row) does *not* appear in the data frame because a variable of interest contains an NA missing value.

Consider the following data:

```
day
animal
food_choice
Monday
eagle
fish
Monday
mountain lion
squirrel
Monday
toad
NA
Tuesday
```

eagle
 fish
 Tuesday
 mountain lion
 deer
 Tuesday
 toad
 flies

Notice that the row for **toad** still appears in the dataset for **Tuesday**, despite having a missing food choice for that day. This is an *explicit missing* because the row still appears in the data frame.

If that row was removed, the resulting dataset would look like this:

```
df_missings %>%
  drop_na(food_choice) %>%
  kable()
```

day
 animal
 food_choice
 Monday
 eagle
 fish
 Monday
 mountain lion
 squirrel
 Tuesday
 eagle
 fish
 Tuesday
 mountain lion
 deer
 Tuesday
 toad

flies

...and if your reaction is “But then how do I know there’s a toad from **MONDAY?**”, then you can see how it can be a bit risky to have *implicit missings* instead of *explicit missings*.

Whichever we choose, we can convert between the two forms using `tidyr::drop_na()` or `tidyr::complete()`:

- `tidyr::drop_na()`: removes observations (rows) that contain NA for variable(s) of interest
- `tidyr::complete()`: turns implicit missing values into explicit missing values by completing a data frame with missing combinations of data

We’ll use both here, starting with the *inverts_long* data frame we created above.

Looking through *inverts_long*, we’ll see that there are NA observations for every species at site **bull** in 2018 - but those NA counts do show up. First, we’ll use `tidyr::drop_na()` to make those missings implicit (invisible) instead:

```
inverts_implicit_NA <- invert_long %>%
  drop_na(sp_count)
```

See that now, the rows that contained an NA in the *sp_count* column from *inverts_long* have been removed.

WAIT, I want them back! We can ask R to create explicit missings (by identifying which combinations of groups currently don’t appear in the data frame) using `tidyr::complete()`:

```
inverts_explicit_NA <- invert_implicit_NA %>%
  complete(month, site, common_name, year)
```

Now you’ll see *inverts_explicit_NA* has those 5 “missing” observations shown in the data frame.

9.4.5 Activities

TODO

9.4.6 Fun facts / insights

TODO

Chapter 10

Dplyr and vlookups

10.1 Summary

In previous sessions, we've learned to do some basic wrangling and find summary information with functions in the `dplyr` package, which exists within the `tidyverse`. We've used:

TODO: Check these to make sure what we do in 1 - 4!

- `dplyr::count()` and `dplyr::uncount()` to get counts of observations for groupings we specify (or the reverse!)
- `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.

10.2 Objectives

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

- Use `dplyr::full_join()`, `dplyr::left_join()`, and `dplyr::inner_join()` 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()` + `dplyr::*_join()` and Excel's VLOOKUP function

10.3 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

10.4 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.

Subset Observations (Rows)



Figure 10.1: `dplyr::filter()`

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

Use `dplyr::filter()` to let R know which **rows** you want to keep or exclude, based whether or not their contents match conditions that you set for one or more variables.

Some examples in words that might inspire you to use `dplyr::filter()`:

- “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).”

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()`.

10.4.1.1 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")
```

10.4.1.2 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 =

10.4.1.3 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
```

10.4.1.4 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")
```

10.4.1.5 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
```

10.4.1.6 Example: combining `filter()` with other functions using the pipe operator (`%>%`)

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

Here, we'll start with the `invert_counts` data frame and create a subset that:

TODO: NOPE, can't do this, they don't learn `pivot_longer()` until the next section (tidying)

- Converts to long format with `pivot_longer()`

- Only keeps observations for rock scallops
- Calculates the total count of rock scallops by site only

```
# Counts of scallops by site (all years included):

scallop_count_by_site <- invert_counts %>%
  pivot_longer(cols = '2016':'2018',
               names_to = "year",
               values_to = "sp_count") %>%
  filter(common_name == "rock scallop") %>%
  group_by(site) %>%
  summarize(tot_count = sum(sp_count, na.rm = TRUE))

scallop_count_by_site
```

```
## # A tibble: 11 x 2
##   site tot_count
##   <chr>      <dbl>
## 1 abur         48
## 2 ahnd         48
## 3 aque        152
## 4 bull         48
## 5 carp       2519
## 6 golb         48
## 7 ivee        169
## 8 mohk        346
## 9 napl       6416
## 10 scdi       2390
## 11 sctw       1259
```

10.4.1.7 Activity 1: using filter() in a wrangling sequence

Write a sequence of code (connected by the pipe operator, %>%), to complete the following and store the output as object `my_fish_wrangling`:

- Starting from the `fish_counts` data frame (stored earlier)
- Only keep observations from *Arroyo Burro* (site 'abur')
- Group by `common_name` (species)
- Find total fish counts (by species) across all years

10.4.2 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.

Combine Data Sets

a	
x1	x2
A	1
B	2
C	3

+

b	
x1	x3
A	T
B	F
D	T

=

Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

dplyr::left_join(a, b, by = "x1")
Join matching rows from b to a.

x1	x3	x2
A	T	1
B	F	2
D	T	NA

dplyr::right_join(a, b, by = "x1")
Join matching rows from a to b.

x1	x2	x3
A	1	T
B	2	F

dplyr::inner_join(a, b, by = "x1")
Join data. Retain only rows in both.

x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

dplyr::full_join(a, b, by = "x1")
Join data. Retain all values, all rows.

Schematic (from RStudio data wrangling cheat sheet):

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.

10.4.2.1 `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:

```
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.

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

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 `dplyr::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")
```

Notice when you look at *fish_kelp_2016_2017*, the 2018 data that **does** exist in *kelp_counts_abur* does **not** get joined to the *fish_2016_2017* data frame,

because `left_join(df_a, df_b)` will only keep observations from `df_b` if they have a match in `df_a`!

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

When we used `left_join(df_a, df_b)`, we kept all observations in `df_a` but *only observations from `df_b` that matched an entry in `df_a`* (in other words, some entries from `df_b` were excluded).

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_fronde
##   <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* in both data frames are returned.

10.4.2.4 dplyr::*_join() in a sequence

We can also merge data frames as part of a sequence of wrangling steps.

As an example: Starting with the `invert_counts` data frame, we want to:

- First, use `pivot_longer()` to get year and counts each into a single column
- Convert the class of `year` to numeric (so it can join with another numeric year variable)
- Then, only keep observations for “california spiny lobster”
- Next, join the `kelp_counts_abur` to the resulting subset above, **only keeping observations that have a match in both data frames**

That might look like this:

```
abur_lobster_kelp <- invert_counts %>%
  pivot_longer('2016':'2018', names_to = "year", values_to = "total_counts") %>%
  mutate(year = as.numeric(year)) %>%
  filter(common_name == "california spiny lobster") %>%
  dplyr::inner_join(kelp_counts_abur)
```

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

```
abur_lobster_kelp
```

```
## # A tibble: 3 x 6
##   month site common_name      year total_counts total_fronds
##   <chr> <chr> <chr>         <dbl>         <dbl>         <dbl>
## 1 7      abur  california spiny lobster 2016             17             307
## 2 7      abur  california spiny lobster 2017             17             604
## 3 7      abur  california spiny lobster 2018             16            3532
```

10.4.2.5 Activity 2

Now let’s combine what we’ve learned about piping, filtering and joining!

Complete the following as part of a single sequence (remember, check to see what you’ve produced after each step) to create a new data frame called `my_fish_join`:

- Start with `fish_counts` data frame
- Filter to only including observations for 2017 at Arroyo Burro
- Join the `kelp_counts_abur` data frame to the resulting subset using `left_join()`
- Add a new column that contains the ‘fish per kelp fronds’ density (`total_count / total_fronds`)

10.5 Fun / kind of scary 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.

10.6 Interludes (deep thoughts/openscapes)

- 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.
- The risk of partial joins (& a case for `full_join + drop_na` instead?)

10.7 Efficiency Tips

- Comment out multiline code with Command + Shift + C
- Knit with Command + Shift + K

Chapter 11

Synthesis

11.1 Summary

In this session, we'll pull together the skills that we've learned so far. We'll create a new GitHub repo and R project, wrangle and visualize data from spreadsheets in R Markdown, communicate between RStudio (locally) and GitHub (remotely) to keep our updates safe, then share our outputs in a nicely formatted GitHub ReadMe. And we'll learn a few new things along the way!

11.2 Objectives

- Create a new repo on GitHub
- Start a new R project, connected to the repo
- Create a new R Markdown document

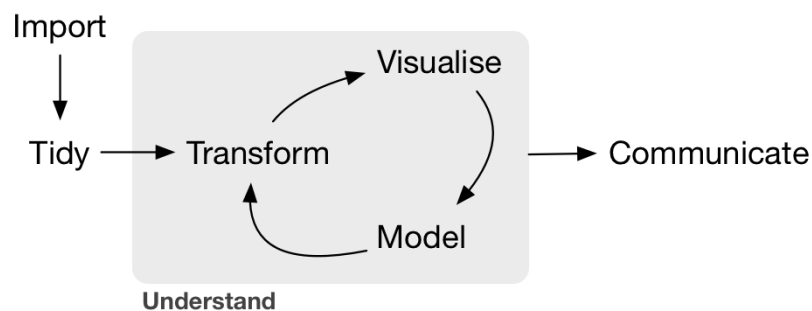


Figure 11.1: Grolemund & Wickham R4DS Illustration

- Attach necessary packages (`googlesheets4`, `tidyverse`, `here`)
- Use `here::here()` for simpler (and safer) file paths
- Read in data from a Google sheet with the `googlesheets4` package in R
- Basic data wrangling (`dplyr`, `tidyr`, etc.)
- Data visualization (`ggplot2`)
- Publish with a useful ReadMe to share

11.3 Resources

- The `here` package
- `googlesheets4` information
- Project oriented workflows by Jenny Bryan

11.4 Lesson

11.4.1 Set-up:

- Log in to your GitHub account and create a new repository called `sea-creature-synthesis`
- Clone the repo to create a version controlled project (remember, copy & paste the URL from the GitHub Clone / Download)
- In the local project folder, create a subfolder called 'data'
- Copy and paste the `fish_counts_curated.csv` and `lobster_counts.csv` into the 'data' subfolder
- Create a new R Markdown document within your `sea-creature-synthesis` project
- Knit your `.Rmd` to html, saving as `sb_sea_creatures.Rmd`

11.4.2 Attach packages and read in the data

Attach (load) packages with `library()`:

```
library(tidyverse)
library(googlesheets4)
library(here)
library(janitor)
```

Now we'll read in our files with `readr::read_csv()`, but our files aren't in our **project root**. They're in the **data** subfolder.

Use `here::here()` to direct R where to look for files, if they're not in the project root. Not sure where that is? Type `here()` in the Console, and it will tell you!

```
here::here()
```

```
"/returns/your/project/root/"
```

Go ahead, find your project root!

Then use `here::here()` *again* to easily locate a file somewhere outside of the exact project root. In our case, the files we want to read in are in the `data` subfolder - so we have to tell R how to get there from the root:

```
# Read in CSV files
```

```
fish_counts <- readr::read_csv(here::here("data", "fish_counts_curated.csv"))
```

```
lobster_counts <- readr::read_csv(here::here("data", "lobster_counts.csv"))
```

Check out the two data frames (`fish_counts` and `lobster_counts`).

The `fish_counts` data frame is in pretty good shape. But the `lobster_counts` df could use some love, because there are “-99999” entries indicating NA values, and the column names would be difficult to write code with.

When reading in the lobster data, let’s:

- convert every “-99999” to an NA
- get the column names into lower snake case using `janitor::clean_names()`

```
lobster_counts <- read_csv(here::here("curation", "lobster_counts.csv"),
                           na = "-99999") %>%
  clean_names()
```

Look at it again to check (always look at your data) - now both data frames seem pretty coder-friendly to work with.

11.4.3 Data wrangling

- join?
- filter?
- unite/separate
- Read in lobster data
- Join with another existing data frame (or 2?)
- Pivoting
- Transforming / subsetting
- Grouping & summarizing (for means, sd, count)
- Make a table
- Make a graph

Possible new things: complete()

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

11.6 Interludes (deep thoughts/openscapes)

11.7 Efficiency Tips