

R for Excel Users

Julie Lowndes & Allison Horst

2019-12-04

Contents

1	Welcome	7
1.1	Prerequisites	8
2	Overview	9
2.1	Why learn R if I know Excel?	9
2.2	Friendly mindset	11
2.3	Guiding principles	11
2.4	Resources	12
3	R & RStudio, RMarkdown	13
3.1	Summary	13
3.2	Objectives	13
3.3	Resources	13
3.4	Why learn R with RStudio	13
3.5	RStudio Orientation	14
3.6	Clearing the environment	21
3.7	Intro to RMarkdown	21
3.8	Deep thoughts	26
3.9	Efficiency Tips	26
4	GitHub	27
4.1	Summary	27
4.2	Objectives	27
4.3	Resources	28
4.4	Why should R users use Github?	29
4.5	Setup Git & GitHub	31
4.6	Create a repository on Github.com	33
4.7	Clone your repository using RStudio	35
4.8	Inspect your repository	39
4.9	Add files to our local repo	40
4.10	Sync from RStudio to GitHub	42
4.11	Explore remote Github	44
4.12	Create a new R Markdown file	45
4.13	Committing - how often? Tracking changes in your files	46

4.14 Happy Git with R	48
4.15 Efficiency Tips	48
5 readxl	49
5.1 Summary	49
5.2 Objectives	49
5.3 Resources	49
5.4 Lesson	50
5.5 Fun facts ideas:	57
5.6 Interludes (deep thoughts/openscapes)	57
5.7 Activity: Import some invertebrates!	57
5.8 Efficiency Tips	58
5.9 Deep thoughts	58
6 Graphs with ggplot2	59
6.1 Summary	59
6.2 Objectives	60
6.3 Resources	61
6.4 Lesson	61
6.5 Activity 1	77
6.6 Activity 2	77
6.7 Efficiency Tips	77
7 Tidying	79
7.1 Summary	79
7.2 Objectives	79
7.3 Resources	79
7.4 Lesson	80
8 Dplyr and vlookups	93
8.1 Summary	93
8.2 Objectives	93
8.3 Resources	94
8.4 Lessons	94
8.5 Fun / kind of scary facts	105
8.6 Interludes (deep thoughts/openscapes)	105
8.7 Efficiency Tips	106
9 Synthesis	107
9.1 Summary	107
9.2 Objectives	107
9.3 Resources	108
9.4 Lesson	108
9.5 Fun facts (quirky things) - making a note of these wherever possible for interest (little “Did you know?” sections)	110
9.6 Interludes (deep thoughts/openscapes)	110

CONTENTS 5

9.7 Efficiency Tips	110
-------------------------------	-----

Chapter 1

Welcome

Hello! This is a workshop taught by Julie Lowndes and Allison Horst at RStudio::conf(2020).

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

Welcome!

We're Julie and Allison. We are environmental scientists who use and teach R in our daily work.

This workshop you will learn hands-on how to begin to interoperate between Excel and R, and develop good habits for working in a reproducible and collaborative way. It's going to be fun and empowering!

We are not only learning R; we will learn a workflow with R and GitHub.

We will practice learning three main things all at the same time: coding with best practices (R/RStudio), collaborative version control (Git/GitHub), and communication/publishing (RMarkdown/GitHub). This training will teach these all together to reinforce skills and best practices, and get you comfortable with a workflow that you can use in your own projects.

2.1 Why learn R if I know Excel?

R and Excel are both great tools that are powerful for different jobs.

Excel is great for data entry. Can also be good for looking at data and feeling like you can touch it, and creating quick exploratory figures.

But Excel can get problematic with extending to analyses. This is because there aren't firm lines between what is data and what is analyses. Also, the analytical steps taken are not readily apparent, nor easy to reproduce. This also makes them pretty brittle/sensitive to minor changes. Have you ever done forensics on an Excel sheet, trying to understand what happened between columns or sheets? Maybe it was even your own Excel file from the (recent) past.

And there are also problems with Excel incorrectly interpreting data as dates, etc.

Has seeing this ever given you a feeling of horror:

R enables you to...

A modern R user has a workflow framed around collaboration, using an ecosystem of tools like GitHub that makes it easier to track versions and share code.

We will learn:

- R, with the RStudio IDE, with the tidyverse
- GitHub

We'll spend the first two sessions getting set up and oriented with R, RMarkdown, and GitHub, and work with data in R all afternoon.

2.1.1 What to expect

This is going to be a fun workshop.

A main theme throughout is to produce analyses people can understand and build from — including Future You and Future Us. The plan is to expose you to tools and workflows 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 are not going to cover everything you know how to do in Excel in R.

But we are 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. Please also ask questions use sticky notes to signal for a teaching assistant.

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.

2.2 Friendly mindset

“pain of failure, it’s just the pain of learning.” - Gordon Shotwell

Something else to start us off is to mention that you are learning a new language here. It’s an ongoing process, it takes time, you’ll make mistakes, it can be frustrating, but it will be overwhelmingly awesome in the long run. We all speak at least one language; it’s a similar process, really. And no matter how fluent you are, you’ll always be learning, you’ll be trying things in new contexts, learning words that mean the same as others, etc, just like everybody else. And just like any form of communication, there will be miscommunications that can be frustrating, but hands down we are all better off because of it.

While language is a familiar concept, programming languages are in a different context from spoken languages, but you will get to know this context with time. For example: you have a concept that there is a first meal of the day, and there is a name for that: in English it’s “breakfast”. So if you’re learning Spanish, you could expect there is a word for this concept of a first meal. (And you’d be right: ‘desayuno’). **We will get you to expect that programming languages also have words (called functions in R) for concepts as well.** You’ll soon expect that there is a way to order values numerically. Or alphabetically. Or search for patterns in text. Or calculate the median. Or reorganize columns to rows. Or subset exactly what you want. We will get you increase your expectations and learn to ask and find what you’re looking for.

You came here to learn R, but we are going to learn R together with RStudio.

2.3 Guiding principles

- “Keep the raw data raw”. A hard line separating raw data and analyses. In R, we have data in one file and written computational commands saved as a separate file.

Write analytical logic in code that can be understood and rerun

Learn from data that are not your own

2.4 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
-
- Spreadsheet Drama (Episode 9) — Not So Standard Deviations with Roger Peng & Hilary Parker

Chapter 3

R & RStudio, RMarkdown

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

In this lesson we will:

- get oriented to the RStudio interface
- work with R in the console
- be introduced to built-in R functions
- learn to use the help pages
- explore RMarkdown

3.3 Resources

- R for Excel Users by Gordon Shotwell (blog)

3.4 Why learn R with RStudio

You are all here today to learn how to code. Coding made me a better scientist because I was able to think more clearly about analyses, and become more

efficient in doing so. Data scientists are creating tools that make coding more intuitive for new coders like us, and there is a wealth of awesome instruction and resources available to learn more and get help.

Here is an analogy to start us off. **Think of yourself as a pilot, and R is your airplane.** You can use R to go places! With practice you'll gain skills and confidence; you can fly further distances and get through tricky situations. You will become an awesome pilot and can fly your plane anywhere.

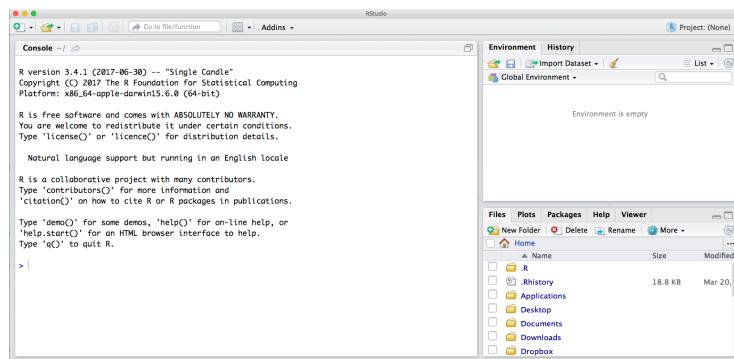
And if R were an airplane, RStudio is the airport. RStudio provides support! Runways, communication, community, and other services that makes your life as a pilot much easier. So it's not only the infrastructure (the user interface or IDE), although it is a great way to learn and interact with your variables, files, and interact directly with GitHub. It's also a data science philosophy, R packages, community, and more. So although you can fly your plane without an airport and we could learn R without RStudio, that's not what we're going to do.

We are learning R together with RStudio and its many supporting features.

3.5 RStudio Orientation

Open RStudio for the first time.

Launch RStudio/R.



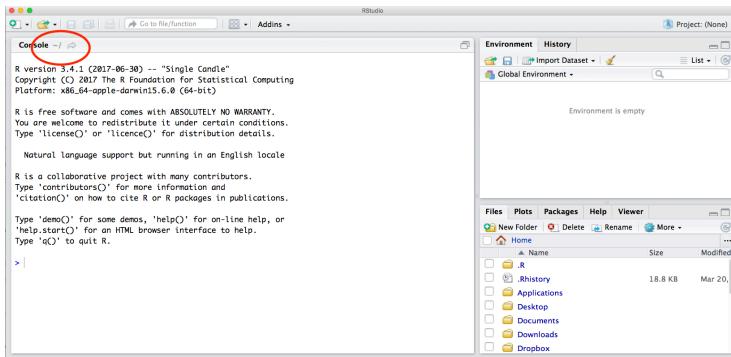
Notice the default panes:

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

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

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

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



3.5.1 R Console

OK let's go into the Console, where we interact with the live R process.

We can do math:

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

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 objects, which are often vectors or dataframes. They are rectangular.

We have to name these dataframes. 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 many people don't do this.)

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

3.5.2 Viewing data in R

Let's have a look at some data in R. Unlike Excel, R comes out-of-the-box with several built-in data sets that we can look at and work with.

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

```
cars
```

This returns data. To me this is not super interesting data, but I can appreciate that there are different variables listed as column headers and then numeric values for each type of row observation. (Unfortunately I don't know if these are different cars or trials or conditions but we won't focus on that for now).

I can also use RStudio's Viewer to see this in a more familiar-looking format. Let's type this — and make sure it's a capital V and open and closed parentheses:

```
View(cars)
```

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.

Aside 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. - Gordon Shotwell

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.

3.5.3 R functions, help pages

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 auto-complete. 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 `cars` data

```
sum(cars)
```

Alright. What is this number? It is the sum of ALL of the data in the `cars` 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 distance:

```
sum(cars$dist)
```

3.5.4 Packages

So far we've been using a couple functions from base R, such as `seq()` and `date()`. But, one of the amazing things about R is that a vast user community is always creating new functions and packages that expand R's capabilities. In R, the fundamental unit of shareable code is the package. A package bundles together code, data, documentation, and tests, and is easy to share with others. They increase the power of R by improving existing base R functionalities, or by adding new ones.

The traditional place to download packages is from CRAN, the Comprehensive R Archive Network, which is where you downloaded R. You can also install packages from GitHub, which we'll do tomorrow.

You don't need to go to CRAN's website to install packages, this can be accomplished within R using the command `install.packages("package-name-in-quotes")`. Let's install a small, fun package `praise`. You need to use quotes around the package name.:

```
install.packages("praise")
```

Now we've installed the package, but we need to tell R that we are going to use the functions within the `praise` package. We do this by using the function `library()`.

What's the difference between a package and a library?

Sometimes there is a confusion between a package and a library, and you can find people calling "libraries" to packages.

Please don't get confused: `library()` is the command used to load a package, and it refers to the place where the package is contained, usually a folder on your computer, while a package is the collection of functions bundled conveniently.

```
library(praise)
```

Now that we've loaded the `praise` package, we can use the single function in the package, `praise()`, which returns a randomized praise to make you feel better.

```
praise()
```

```
## [1] "You are super-duper!"
```

3.5.5 Assigning objects with <-

This might be a really important value that we want to have on hand. We can save this as its own object. We do this by writing the name along with the assignment operator <-

```
sum_dist <- sum(cars$dist)
```

And we can execute it. In my head I hear “sum_dist gets 2149”.

Object names can be whatever you want, although they cannot start with a digit and cannot contain certain other characters such as a comma or a space. Different folks have different conventions; you will be wise to adopt a convention for demarcating words in names.

```
# i_use_snake_case
# other.people.use.periods
# evenOthersUseCamelCase
```

Notice that as I start typing `sum_dist` in the Console, there will be options to auto-fill. This is RStudio helping you out, which is great because we all are prone to typos. I actually have to ignore the help to try to force a typo:

```
sumdist
# Error: object 'sumdist' not found
```

OK this is an error, but I didn’t break R — error messages are your friends.

The first thing to do with an error message is read it. Yes it’s in angry red text and it’s unexpected — but most error messages are doing their best to help you solve the problem. And you’ll get more familiar with them the way they tell you. By saying “object ‘sumdist’ not found” alerts me immediately to the fact that this thing I think exists R doesn’t think exists — so maybe it’s a typo or not loaded?

3.5.6 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!

3.5.7 Logical operators and expressions

A moment about **logical operators and expressions**. We can ask questions about the objects we made.

- `==` means ‘is equal to’
- `!=` means ‘is not equal to’
- `<` means ‘is less than’
- `>` means ‘is greater than’
- `<=` means ‘is less than or equal to’
- `>=` means ‘is greater than or equal to’

```
sum_dist == 2
## [1] FALSE
sum_dist <= 3000
## [1] TRUE
sum_dist != 500
## [1] TRUE
```

Shortcuts You will make lots of assignments and the operator `<-` is a pain to type. Don’t be lazy and use `=`, although it would work, because it will just sow confusion later. Instead, utilize **RStudio’s keyboard shortcut: Alt + - (the minus sign)**. Notice that RStudio automagically surrounds `<-` with spaces, which demonstrates a useful code formatting practice. Code is miserable to read on a good day. Give your eyes a break and use spaces. RStudio offers many handy keyboard shortcuts. Also, Alt+Shift+K brings up a keyboard shortcut reference card.

My most common shortcuts include command-Z (undo), and combinations of arrow keys in combination with shift/option/command (moving quickly up, down, sideways, with or without highlighting).

3.6 Clearing the environment

Now look at the objects in your environment (workspace) – in the upper right pane. The workspace is where user-defined objects accumulate.

For reproducibility, it is critical that you delete your objects and restart your R session frequently. You don't want your whole analysis to only work in whatever way you've been working right now — you need it to work next week, after you upgrade your operating system, etc. Restarting your R session will help you identify and account for anything you need for your analysis.

We will keep coming back to this theme but let's restart our R session together: Go to the top menus: Session > Restart R.

Don't save the workspace!

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.

3.7 Intro to RMarkdown

3.7.1 What is RMarkdown? (1-minute video)

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

What is RMarkdown?

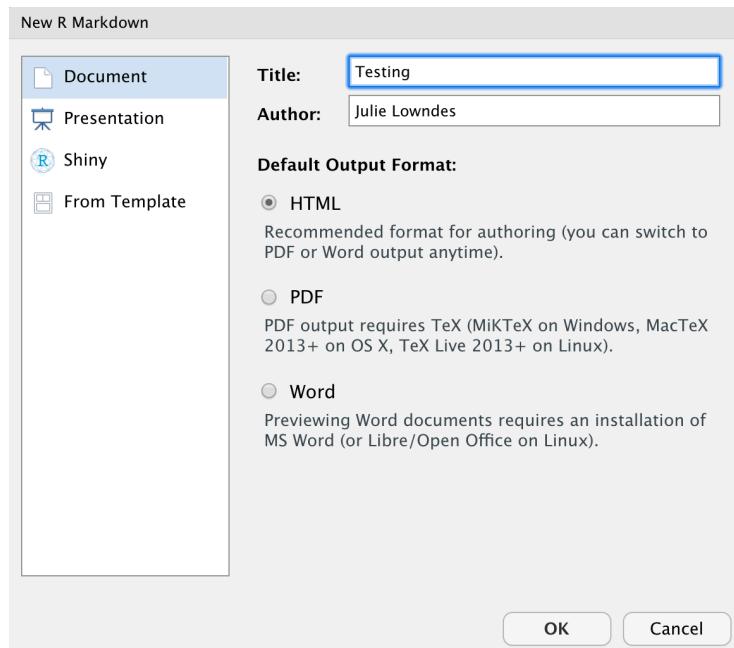
An RMarkdown file will allow us to weave markdown text with chunks of R code to be evaluated and output content like tables and plots.

This is really critical to reproducibility, and it also saves time. This document will recreate your figures for you in the same document where you are writing text. So no more doing analysis, saving a plot, pasting that plot into Word, redoing the analysis, re-saving, re-pasting, etc.

Let's experience this a bit ourselves and then we'll talk about it more.

3.7.2 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.
34

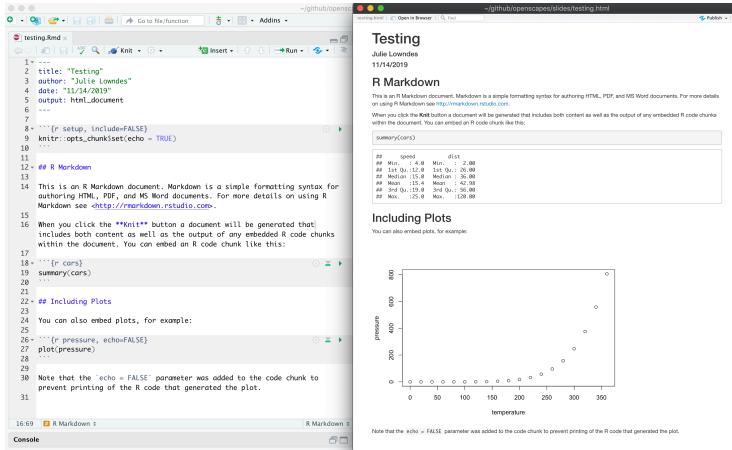
```

3.7.3 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:



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

3.7.4 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.

3.7.5 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`.

We can 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 code in R. Earlier we calculated `sum(cars$dist)`. Now let's do the average. In R, this is the `mean()` function

```
mean_dist <- mean(cars$dist)
```

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

And when we do this, we see the `mean_dist` object appear in he Environment pane.

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

3.7.6 Activity (3 min)

Add a few more commands to your file from this morning. Execute your commands by trying the three ways above. Then, knit your R Markdown file, which will also save the Rmd by default. Remember to write your R code within a code chunk (grey)!

3.7.7 Markdown sections

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 in a few ways in real life: it can mean “number” or “pound” or hashtags on social media.

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

3.7.8 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.

3.7.9 Restart R

To end our work from this session, save, knit, and then restart R (Go to the top menus: Session > Restart R.)

Notice that now with a clean workspace, if I knit my document instead of sending code to the Console, my objects (like `mean_dist`) don't show up in my Environment. This is because R isn't actually running this in this R session, it is actually spinning up a clean session to knit my document. This is important for reproducible analyses because I don't want the success of this analysis to be dependent on some weird setting I have on my computer that will make Future Me or Future Us not able to run or understand these important analyses. Having RMarkdown be self-contained in this way helps you develop good habits for reproducibility.

3.8 Deep thoughts

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 Efficiency Tips

—>

Chapter 4

GitHub

TO ADD:

- create data/ folder. (GH won't see an empty folder so put a README in it).

I like to turn off workbook mode.

Create the github repo: ‘r-and-excel’

4.1 Summary

We will learn about version control using git and GitHub, and we will interface with this through RStudio. Why use version control? To save time when working with your most important collaborator: you.

4.2 Objectives

Today, we'll interface with GitHub from our local computers using RStudio. There are many other ways to interact with GitHub, including GitHub's Desktop App or the command line (here is Jenny Bryan's list of git clients), but today we are going to work from RStudio. You have the largest suite of options if you interface through the command line, but the most common things you'll do can be done through one of these other applications (i.e. RStudio and the GitHub Desktop App).

Here's what we'll do after we set up git on your computers:

1. create a repository on Github.com
2. clone locally using RStudio

3. learn the RStudio-GitHub workflow by syncing to Github.com: pull, stage, commit, push
4. explore github.com: files, commit history, file history
5. practice the RStudio-GitHub workflow by editing and adding files
6. practice R Markdown

git will track and version your files, GitHub stores this online and enables you to collaborate with others (and yourself). Although git and GitHub are two different things, distinct from each other, we can think of them as a bundle since we will always use them together. It also helped me to think of GitHub like Dropbox: you make folders that are ‘tracked’ and can be synced to the cloud. GitHub does this too, but you have to be more deliberate about when syncs are made. This is because GitHub saves these as different versions, with information about who contributed when, line-by-line. This makes collaboration easier, and it allows you to roll-back to different versions or contribute to others’ work.

4.3 Resources

These materials borrow from:

- Jenny Bryan’s lectures from STAT545 at UBC: The Shell
- Jenny Bryan’s Happy git with R tutorial
- Melanie Frazier’s GitHub Quickstart, GitHub Lesson at University of Queensland
- Ben Best’s Software Carpentry at UCSB

Today, we’ll only introduce the features and terminology that scientists need to learn to begin managing their projects.

4.4 Why should R users use Github?

1. Ends (or, nearly ends) the horror of keeping track of versions. Basically, we

<input type="checkbox"/> Name	Date modified	Type
Rscript_4_21_2016.R	5/1/2016 3:03 PM	R File
Rscript_4_22_2016a.R	5/1/2016 3:03 PM	R File
Rscript_4_22_2016b.R	5/1/2016 3:03 PM	R File
Rscript_4_24_2016.R	5/1/2016 3:03 PM	R File
Rscript_final.R	5/1/2016 3:03 PM	R File
Rscript_final_final.R	5/1/2016 3:03 PM	R File
Rscript_really_final.R	5/1/2016 3:03 PM	R File
Rscript_really_really_final.R	5/1/2016 3:03 PM	R File

get away from this:

When you open your repository, you only see the most recent version. But, it's easy to compare versions, and you can easily revert to previous versions.

2. Improves collaborative efforts. Different researchers can work on the same files at the same time!
3. It is easy to share and distribute files through the Github website.
4. Your files are available anywhere, you just need internet connection!

4.4.1 What are Git and Github?

- **Git** is a version control system that lets you track changes to files over time. These files can be any kind of file (eg .doc, .pdf, .xls), but free text differences are most easily visible (eg txt, csv, md).
- **Github** is a website for storing your git versioned files remotely. It has many nice features to be able to visualize differences between images, rendering & diffing map data files, render text data files, and track changes in text.

If you are a student you can get the micro account which includes 5 private repositories for free (normally a \$7/month value). You can sign up for the student account here. Instructors can also request a free organization account, “Request a discount”.

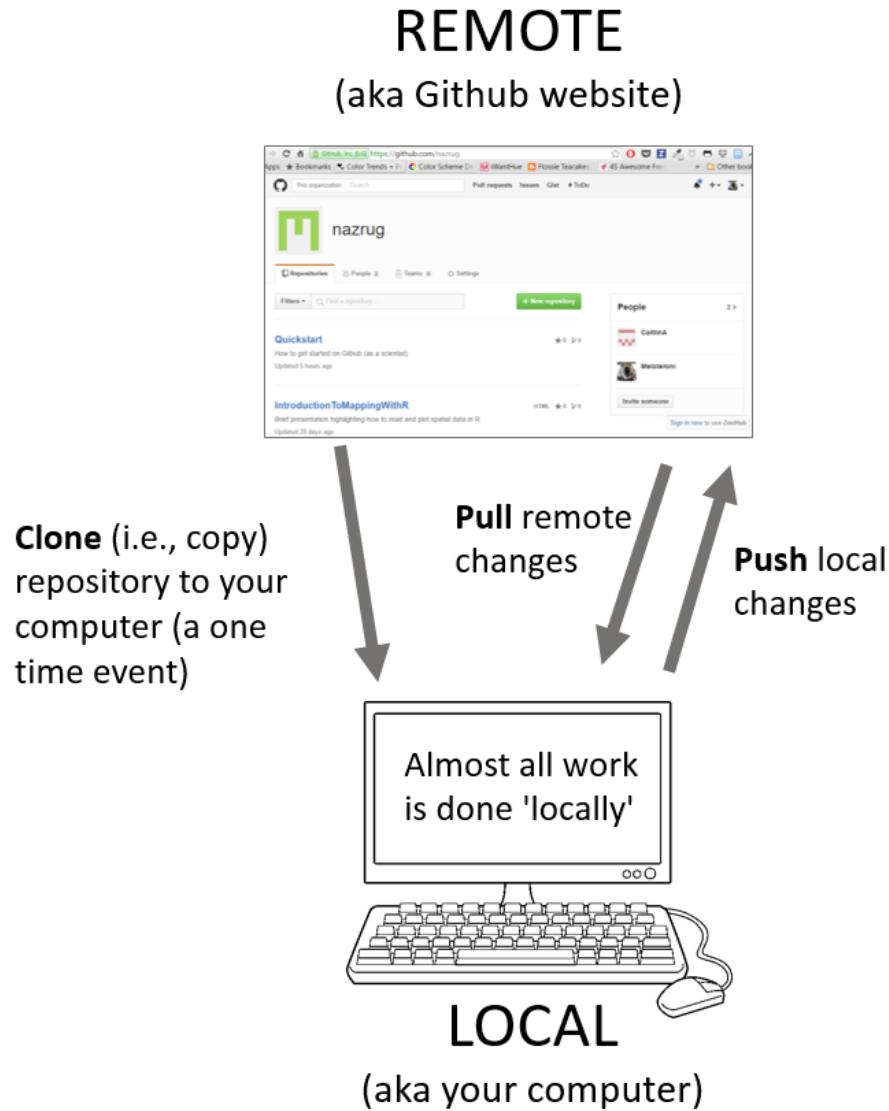
Github was developed for social coding (i.e., sort of like an open source Wikipedia for programmers). Consequently, much of the functionality and terminology of Github (e.g., branches and pull requests) isn't necessary for a scientist getting started.

These concepts are more important for coders who want the entire coding community (and not just people working on the same project) to be able to suggest changes to their code. This isn't how most scientists will use Github.

To get the full functionality of Github, you will eventually want to learn other concepts. But, this can wait.

4.4.2 Some Github terminology

- **User:** A Github account for you (e.g., jules32).
- **Organization:** The Github account for one or more user (e.g., datacar-pentry).
- **Repository:** A folder within the organization that includes files dedicated to a project.
- **Local Github:** Copies of Github files located your computer.
- **Remote Github:** Github files located on the <https://github.com> website.
- **Clone:** Process of making a local copy of a remote Github repository. This only needs to be done once (unless you mess up your local copy).
- **Pull:** Copy changes on the remote Github repository to your local Github repository. This is useful if multiple people are making changes to a repository.
- **Push:** Save local changes to remote Github



4.5 Setup Git & GitHub

We're going to switch gears from R for a moment and set up Git and GitHub, which we will be using along with R and RStudio for the rest of the workshop. This set up is a one-time thing! You will only have to do this once per computer. We'll walk through this together.

1. Create **Github** account at <http://github.com>, if you don't already have one. For username, I recommend all lower-case letters, short as you can.

I recommend using your *.edu email*, since you can request free private repositories via GitHub Education discount.

2. You will use the `usethis` package to configure `git` with global commands, which means it will apply ‘globally’ to all files on your computer, rather than to a specific folder.

```
install.packages("usethis")
library(usethis)

use_git_config(user.name = "Melsteroni", user.email = "Melsteroni@example.org")
```

BACKUP PLAN If `usethis` fails, the following is the classic approach to configuring `git`. Open the Git Bash program (Windows) or the Terminal (Mac) and type the following:

```
# display your version of git
git --version

# replace USER with your Github user account
git config --global user.name USER

# replace NAME@EMAIL.EDU with the email you used to register with Github
git config --global user.email NAME@EMAIL.EDU

# list your config to confirm user.* variables set
git config --list
```

Not only have you just set up `git` as a one-time-only thing, you have just used the command line. We don’t have time to learn much of the command line today, but you just successfully used it following explicit instructions, which is huge! There are great resources for learning the command line, check out this tutorial from SWC at UCSB.

4.5.1 Troubleshooting

If you have problems setting up `git`, please see the Troubleshooting section in Jenny Bryan’s amazing HappyGitWithR.

4.5.1.1 New(ish) Error on a Mac

We’ve also seen the following errors from RStudio:

```
error key does not contain a section --global terminal
and
fatal: not in a git directory
```

To solve this, go to the Terminal and type: `which git`

Look at the filepath that is returned. Does it say anything to do with Apple?

-> If yes, then the Git you downloaded isn't installed, please redownload if necessary, and follow instructions to install.

-> If no, (in the example image, the filepath does not say anything with Apple) then proceed below:

In RStudio, navigate to: Tools > Global Options > Git/SVN.

Does the “**Git executable**” filepath match what the url in Terminal says?

If not, click the browse button and navigate there.

Note: on my laptop, even though I navigated to /usr/local/bin/git, it then automatically redirect because /usr/local/bin/git was an alias on my computer. That is fine. Click OK.

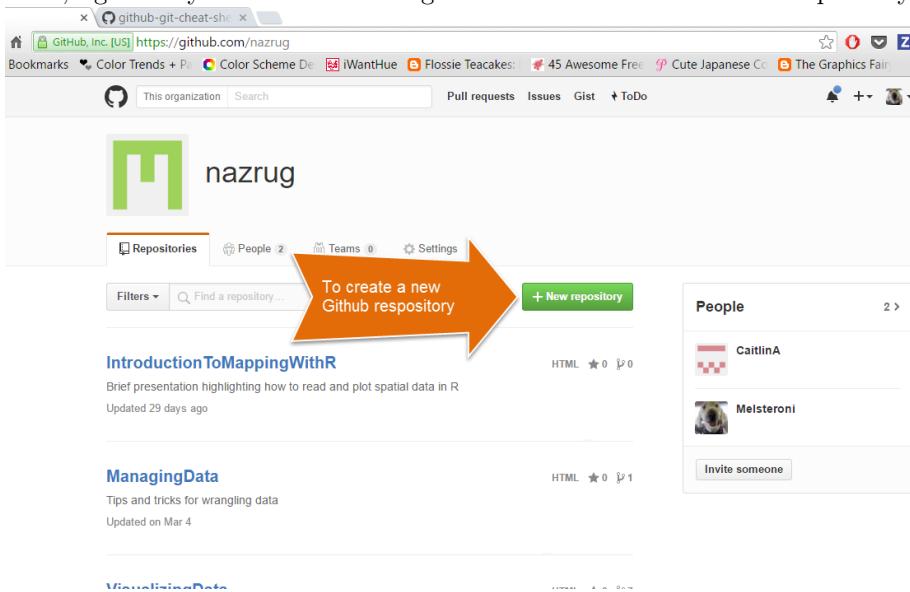
Quit RStudio.

Then relaunch RStudio.

Try syncing or cloning, and if that works and then you don't need to worry about typing into the Terminal, you're all done!

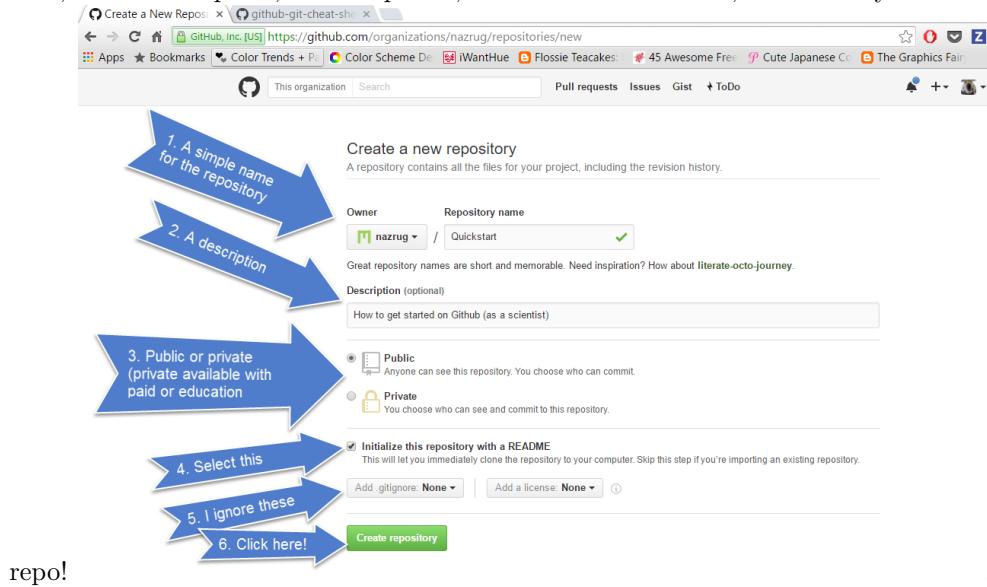
4.6 Create a repository on Github.com

First, go to your account on github.com and click “New repository”.



Choose a name. Call it whatever you want (the shorter the better), or follow me for convenience. I will call mine `my-repo`.

Also, add a description, make it public, create a README file, and create your

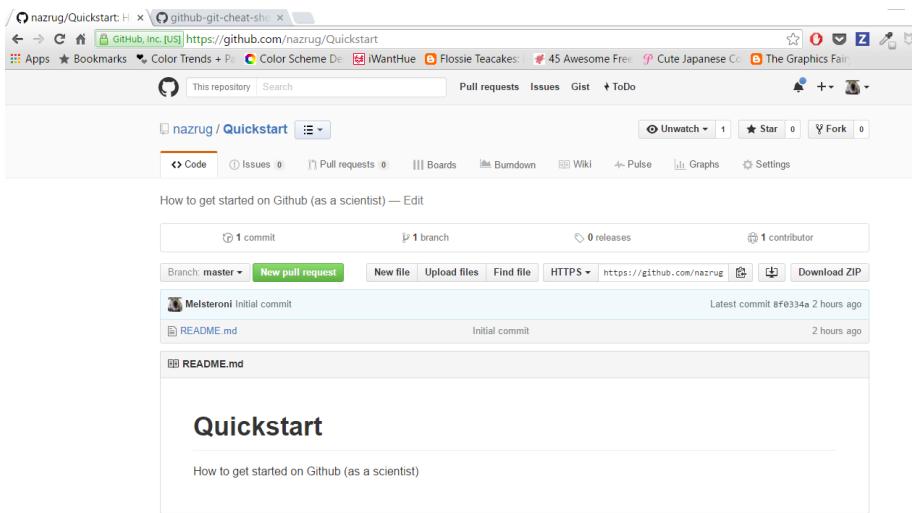


The *Add gitignore* option adds a document where you can identify files or file types you want Github to ignore. These files will stay in on the local Github folder (the one on your computer), but will not be uploaded onto the web version of Github.

The *Add a license* option adds a license that describes how other people can use your Github files (e.g., open source, but no one can profit from them, etc.). We won't worry about this today.

Check out our new repository!

Notice how the `README.md` file we created is automatically displayed at the bottom. The `.md` means that it is Markdown (remember how `.Rmd` was RMarkdown?) so the formatting we learned in the last lesson apply here.



From here, you will work locally (on your computer).

4.7 Clone your repository using RStudio

We'll start off by cloning to our local computer using RStudio. We are going to be cloning a copy of our Remote repository on GitHub.com to our local computers. Unlike downloading, cloning keeps all the version control and user information bundled with the files.

Step 0: Create your `github` folder

This is really important! We need to be organized and deliberate about where we want to keep all of our GitHub repositories (since this is the first of many in your career).

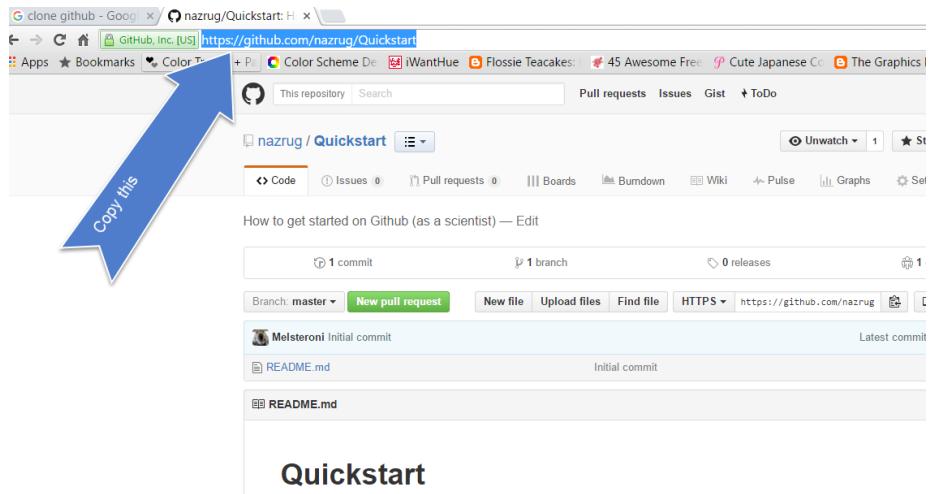
Let's all make a folder called `github` (all lowercase!) in our home directories. So it will look like this:

- Windows: `Users\[User]\Documents\github\`
- Mac: `Users/[User]/github/`

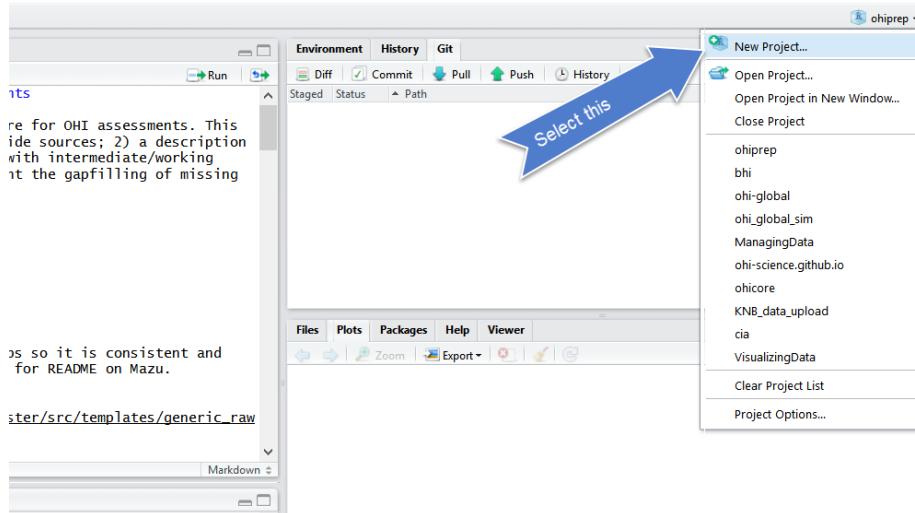
This will let us take advantage of something that is really key about GitHub.com: you can easily navigate through folders within repositories and the urls reflect this navigation. The greatness of this will be evident soon. So let's set ourselves up for easily translating (and remembering) those navigation paths by having a folder called `github` that will serve as our 'github.com'.

So really. Make sure that you have an all-lowercase folder called `github` in your home directory!!

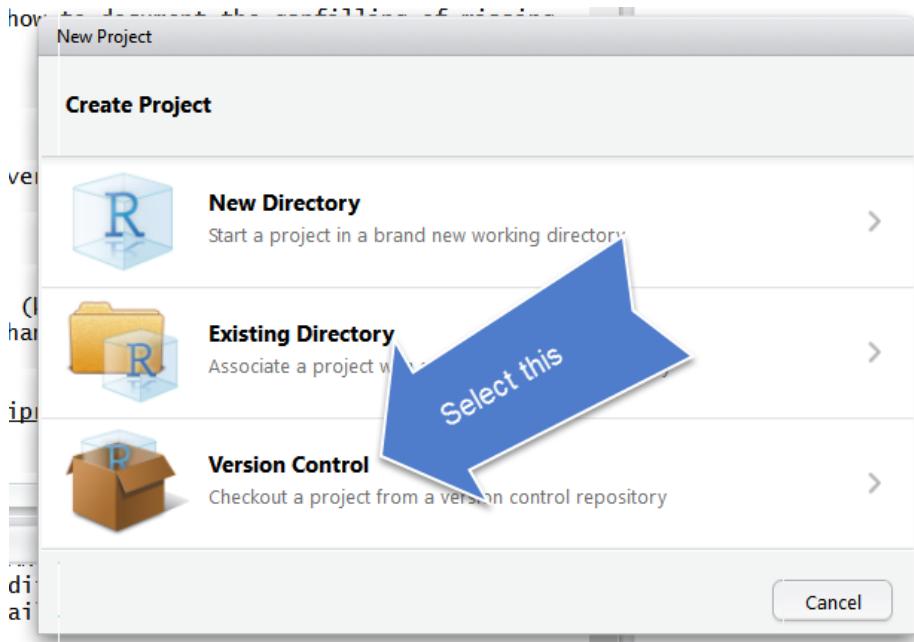
Step 1: Copy the web address of the repository you want to clone.



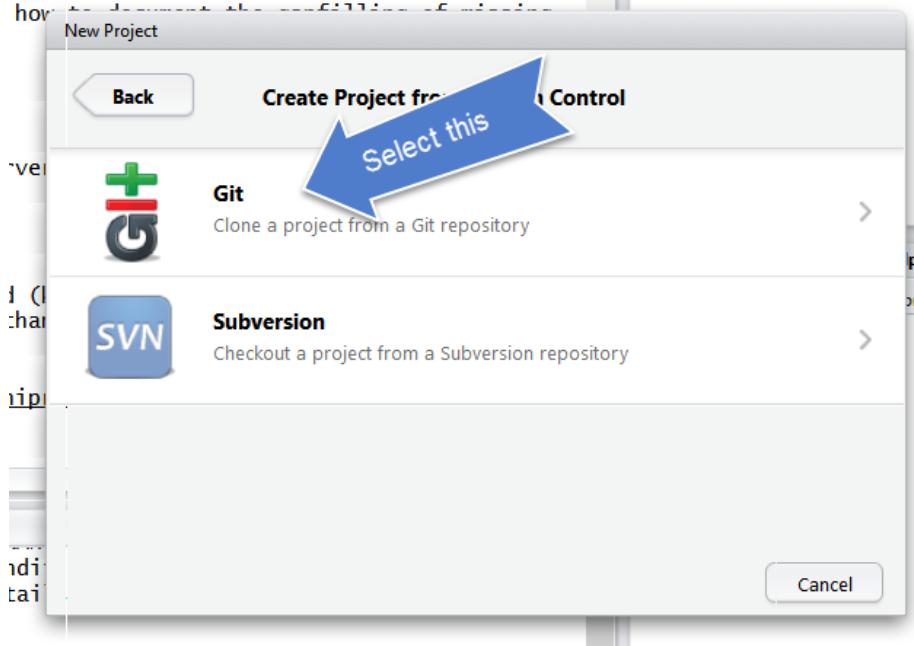
Step 2: from RStudio, go to New Project (also in the File menu).



Step 3: Select Version Control



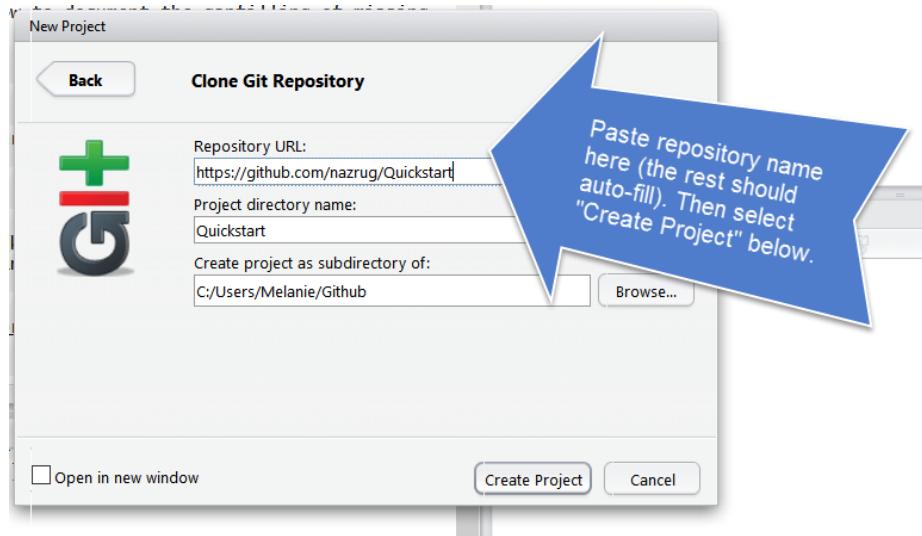
Step 4: Select Git



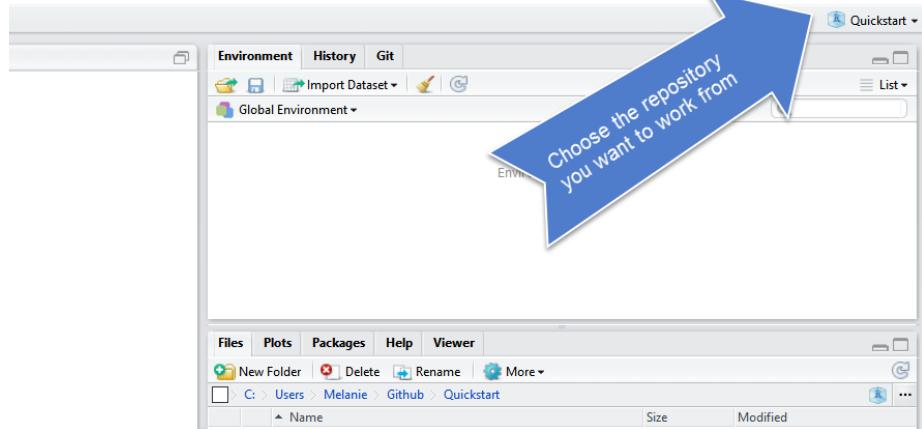
Step 5: Paste it in the Repository URL field, and type **tab** to autofill the Project Directory name. Make sure you keep the Project Directory Name **THE**

SAME as the repository name from the URL.

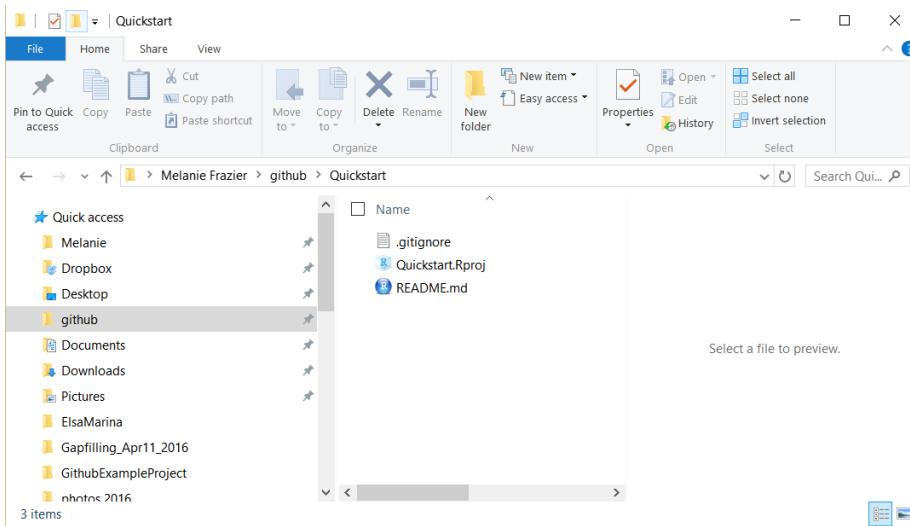
Save it in your github folder (click on Browse) to do this.



If everything went well, the repository will be added to the list located here:



And the repository will be saved to the Github folder on your computer:

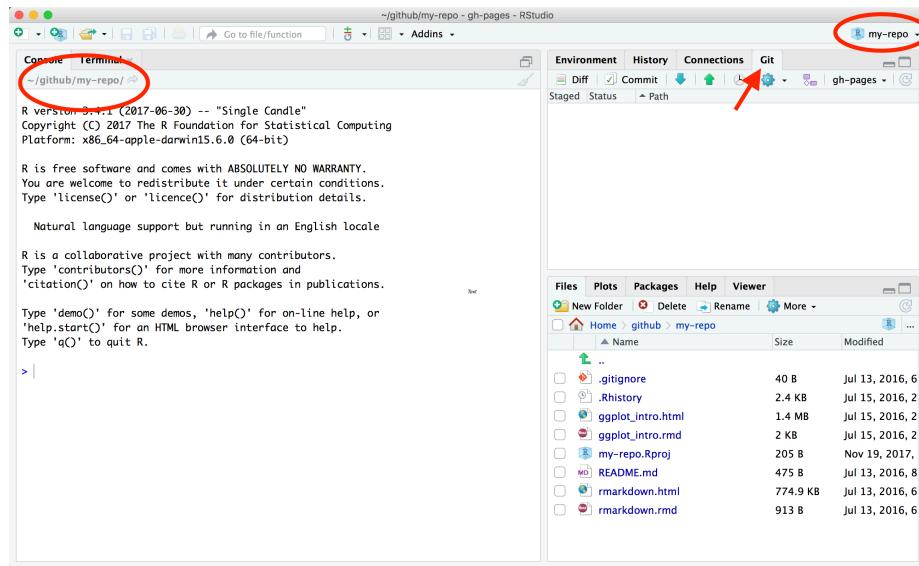


Ta da!!!! The folder doesn't contain much of interest, but we are going to change that.

4.8 Inspect your repository

Notice a few things in our repo here:

1. Our working directory is set to `~/github/my-repo`. This means that I can start working with the files I have in here without setting the filepath. This is that when we cloned this from RStudio, it created an RStudio project, which you can tell because:
 - `.RProj` file, which you can see in the Files pane.
 - The project is named in the top right hand corner
2. We have a git tab! This is how we will interface directly to Github.com



When you first clone a repo through RStudio, RStudio will add an `.Rproj` file to your repo. And if you didn't add a `.gitignore` file when you originally created the repo on GitHub.com, RStudio will also add this for you. These will show up with little yellow ? icons in your git tab. This is GitHub's way of saying: "I am responsible for tracking everything that happens in this repo, but I haven't seen these files yet. Do you want me to track them too?" (We'll see that when you click the box to stage them, they will turn into As because they have been added to the repo).

4.9 Add files to our local repo

The repository will contain:

- `.gitignore` file
- `README.md`
- `Rproj`

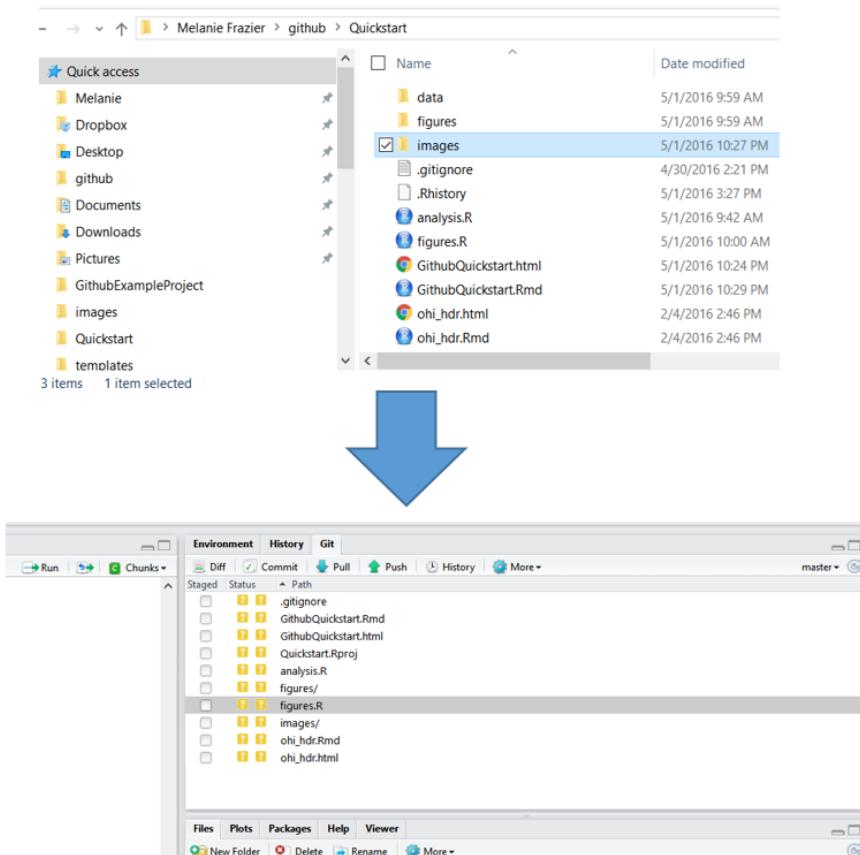
And, I typically create the following:

- folders for “data” and “figures”
- R scripts
- etc.

I'm going to go to the Finder (Windows Explorer on a PC) and copy a file into my repository from there. And then I'm going to go back to RStudio – it shows up in the git tab! So the repository is being tracked, no matter how you make changes to it (changes do not have to be done only through RStudio).

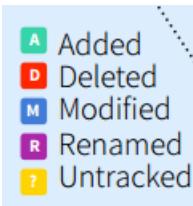
To make changes to the repository, you will work from your computer (“local Github”).

When files are changed in the local repository, these changes will be reflected in the Git tab of RStudio:



4.9.1 Inspect what has changed

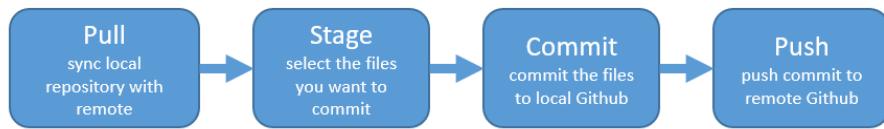
These are the codes RStudio uses to describe how the files are changed, (from



the RStudio cheatsheet):

4.10 Sync from RStudio to GitHub

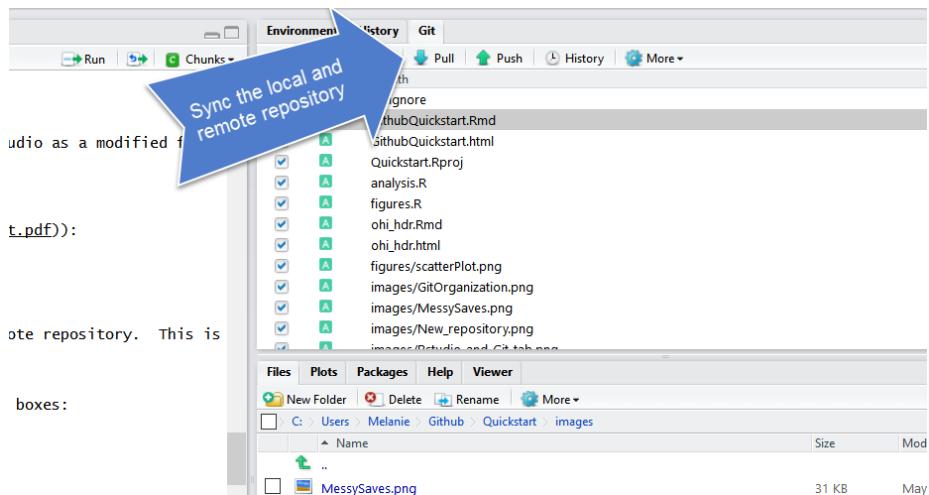
When you are ready to commit your changes, you follow these steps:



We walk through this process below:

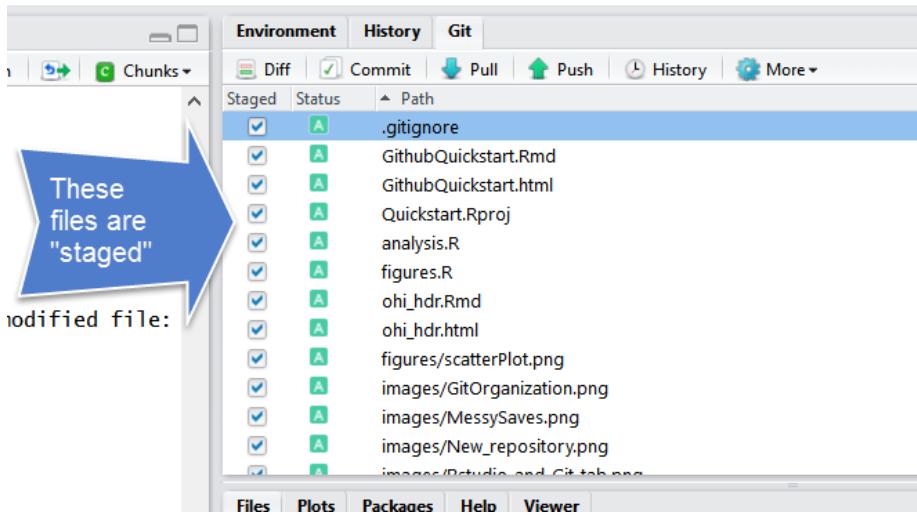
4.10.1 Pull

From the Git tab, “Pull” the repository. This makes sure your local repository is synced with the remote repository. This is very important if other people are making changes to the repository or if you are working from multiple computers.

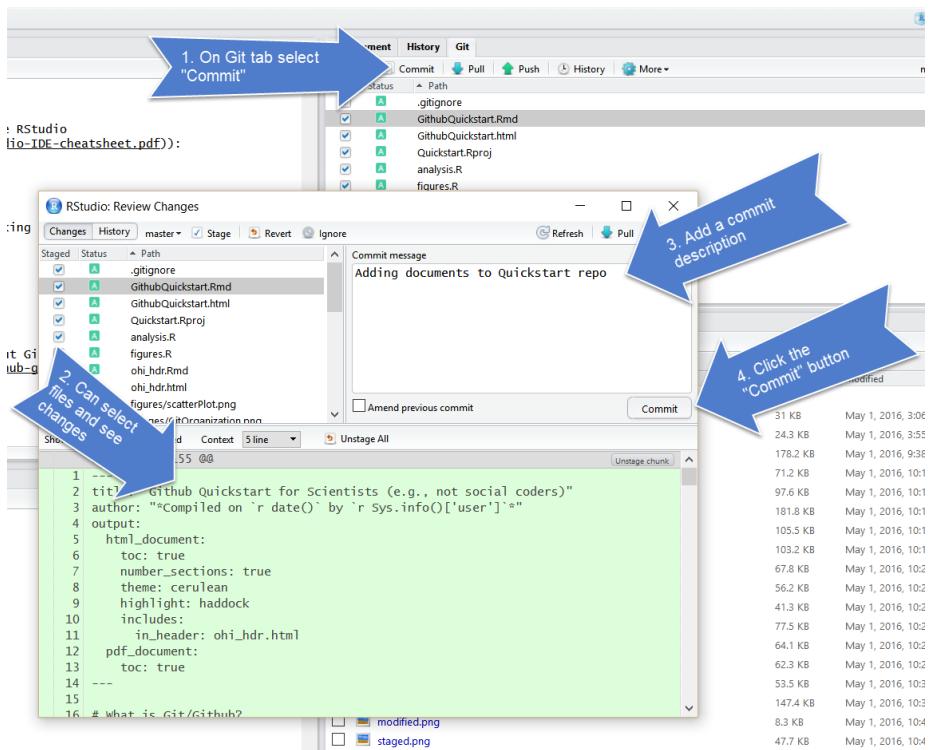


4.10.2 Stage

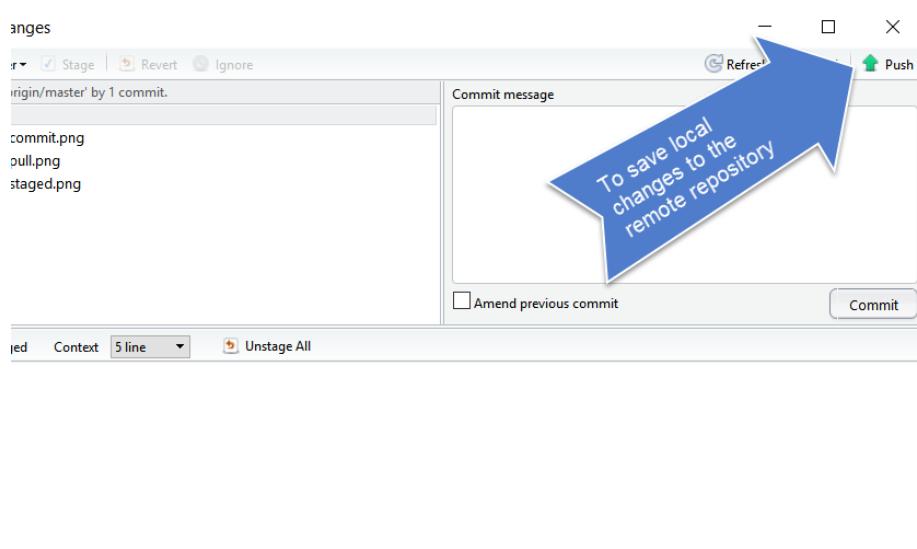
Stage the files you want to commit. In RStudio, this involves checking the “Staged” boxes:



4.10.3 Commit



4.10.4 Push



4.11 Explore remote Github

The files you added should be on github.com:

The screenshot shows a GitHub repository page for 'nazrug / Quickstart'. At the top, there are various links like 'Color Trends', 'Color Scheme', 'iWantHue', 'Flossie Teacakes', '45 Awesome Free', 'Cute Japanese CC', 'The Graphics Fairy', and 'Backstitch sewin'. Below the header, the repository name 'nazrug / Quickstart' is displayed along with a dropdown menu, search bar, and navigation links for 'Pull requests', 'Issues', 'Gist', 'ToDo', 'Unwatch', 'Star', 'Fork', and 'Settings'. The main content area shows the commit history:

Commit	Message	Time Ago
Latest commit cad70a4	Adding Quickstart files	21 minutes ago
figures	Adding Quickstart files	21 minutes ago
images	Adding Quickstart files	21 minutes ago
.gitignore	Adding Quickstart files	21 minutes ago
GithubQuickstart.Rmd	Adding Quickstart files	21 minutes ago
GithubQuickstart.html	Adding Quickstart files	21 minutes ago
Quickstart.Rproj	Adding Quickstart files	21 minutes ago
README.md	Initial commit	2 days ago
analysis.R	Adding Quickstart files	21 minutes ago
figures.R	Adding Quickstart files	21 minutes ago
ohi_hdr.Rmd	Adding Quickstart files	21 minutes ago
ohi_hdr.html	Adding Quickstart files	21 minutes ago

Let's also explore commit history, file history.

4.11.1 Activity

This time let's edit an existing file instead of adding something new. Open your README file by clicking on it in the Files pane (lower right corner). Write a few lines of text, save, and see what happens in your Git Tab. Sync it to your remote repository (Github.com).

Also, go to your Finder/Windows Explorer, and copy-paste something into your local GitHub repo. Then go back to RStudio and confirm that git tracked it. Remember, git will track anything within that folder (the way Dropbox does), it's not specific to RStudio!

4.12 Create a new R Markdown file

OK, now, let's go back to RStudio, and get ourselves back into learning R. We are going to use R Markdown so that you can write notes to yourself in Markdown, and have a record of all your R code. Writing R commands in the console like we did this morning is great, but limited; it's hard to keep track of and hard to efficiently share with others. Plus, as your analyses get more complicated, you need to be able to see them all in one place.

Go to File > New File > R Markdown ... (or click the green plus in the top left corner).

Let's set up this file so we can use it for the rest of the day. I'm going to delete all the text that is already there and write some new text.

Here's what I'm going to write in my R Markdown file to begin:

```
---
title: "Reading data into R with `readxl`"
author: "Julie Lowndes"
date: "12/7/2019"
output: html_document
---

# Learning `readxl`

We are working with data and it's going to be amazing.
```

Now, let's save it. I'm going to call my file `readxl.Rmd`. You can knit it if you'd like.

Then, sync your file to GitHub.

What if a file doesn't show up in the Git tab and you expect that it should? Check to make sure you've saved the file. If the filename is red with an asterix, there have been changes since it was saved. Remember to save before syncing to GitHub!

4.13 Committing - how often? Tracking changes in your files

Whenever you make changes to the files in Github, you will walk through the Pull -> Stage -> Commit -> Push steps.

I tend to do this every time I finish a task (basically when I start getting nervous that I will lose my work). Once something is committed, it is very difficult to lose it.

One thing that I love about Github is that it is easy to see how files have changed over time. Usually I compare commits through github.com:

4.13. COMMITTING - HOW OFTEN? TRACKING CHANGES IN YOUR FILES47

GitHub, Inc. [US] https://github.com/OHI-Science/ohiprep

This repository Search Pull requests Issues Gist + ToDo

OHI-Science / ohiprep

Code Issues Pull requests Boards Burndown Wiki Pulse Graphs Settings

Green Health Index data layer preparation — Edit

1,662 commits 2 branches 1 release 8 contributors

Branch: master New pull request New file Upload files Find file HTTPS https://github.com/OHI-Sc Fork Download ZIP

Latest commit sf#aba9 7 hours ago

Author	Message	Time
Antarctica	Updating ICDs for AQ	5 days ago
Baltic/StockholmUniversity-Regions_v...	updated regions, buffers. about to try on optimus.	2 years ago
China/ChinaRegions/data	Downloading new ohiprep to my computer	2 years ago
Global	AO need gapfilling	2 months ago
HighSeas	Changing paths on AQ RES RFMO	2 years ago
Israel	removing vestigial code from the Israel resilience script	2 years ago
Reference	LSP: correcting resilience files	a month ago
globalprep	debugged and reprocessed the SPP goal... done with that? now to make ...	7 hours ago
sac	Create README.md	4 days ago

This repository Search Pull requests Issues Gist + ToDo

OHI-Science / ohiprep

Code Issues Pull requests Boards Burndown Wiki Pulse Graphs

Branch: master

Commits on May 2, 2016

debugged and reprocessed the SPP goal... done with that? now to make ... = committed 7 hours ago

Commits on May 1, 2016

Merge branch 'master' of https://github.com/OHI-Science/ohiprep = committed a day ago

debugging the species for 2016... I think it's actually OK... = committed a day ago

Commits on Apr 28, 2016

Create README.md ningning committed 4 days ago

data organization SOP revisions Melsteroni committed 4 days ago

Commits on Apr 27, 2016

You can click on the commits to see how the files changed from the previous commit:

The screenshot shows a GitHub commit page for the repository 'OHI-Science / ohiprep'. The commit message is 'debugging the species for 2016... I think it's actually OK...'. The commit was made a day ago by a user (@) and has 1 parent, commit b21fe2946589b4b2ed7351bc68c154651aeb7953. The commit shows 88 changed files with 2,432,873 additions and 426,265 deletions. A note says 'Sorry, we could not display the entire diff because it was too big.' The diff itself shows a script named 'globalprep/spp_ico/2016/data_prep_SPP.Rmd'. The code changes are color-coded: red for deleted lines and green for added lines. Two large blue arrows point from the text 'Code in red was deleted' and 'Code in green was added' towards the respective colored regions in the diff view.

```

@@ -22,18 +22,18 @@ library(foreign)
library(data.table)
library(sp)
library(rgdal)
-library(raster)
-library(maptools)
+library(raster)
+library(maptools)

library(readr)

source('~/github/ohiprep/src/R/common.R')

goal <- 'globalprep/spp_ico'
scenario <- '2016'
dir_anx <- file.path(dir_M, 'git-anne', goal)
dir_data_am <- file.path(dir_M, 'git-anne/globalprep/_raw_data', 'aquamaps', str_replace(scenario, 'v', 'd'))
dir_data_luc <- file.path(dir_M, 'git-anne/globalprep/_raw_data', 'luc spp')
dir_data_bird <- file.path(dir_M, 'git-anne/globalprep/_raw_data', 'birdlife.intl')
- dir_data_am <- file.path(dir_M, 'git-anne/globalprep/_raw_data', 'aquamaps', d2015)
+ dir_data_am <- file.path(dir_M, 'git-anne/globalprep/_raw_data', 'aquamaps', d2015)
- dir_data_luc <- file.path(dir_M, 'git-anne/globalprep/_raw_data', 'luc spp', d2015)
+ dir_data_luc <- file.path(dir_M, 'git-anne/globalprep/_raw_data', 'luc spp', d2015)
- dir_data_bird <- file.path(dir_M, 'git-anne/globalprep/_raw_data', 'birdlife.intl', d2015)
+ dir_data_bird <- file.path(dir_M, 'git-anne/globalprep/_raw_data', 'birdlife.intl', d2015)
dir_git <- file.path('~/github/ohiprep', goal)

```

4.14 Happy Git with R

If you have problems, we'll help you out using Jenny Bryan's HappyGitWithR, particularly the sections on Detect Git from RStudio and RStudio, Git, GitHub Hell (troubleshooting). So as we are coming around, have a look at it and see if you can help troubleshoot too!

4.15 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 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 7 files for this workshop, some comma separate value (CSV) files and some as Excel spreadsheets (.xlsx):

- `fish_counts_curated.csv`
- `invert_counts_curated.xlsx`
- `kelp_counts_curated.xlsx`
- `substrate_cover_curated.xlsx`
- `lobster_counts.csv`
- `ca_np.csv`
- `ci_np.xlsx`

Copy and paste those files into the ‘data’ subfolder of your R project. Notice that now these files are in your working directory when you go back to that Project in RStudio (check the ‘Files’ tab and navigate to the data subfolder). 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, writexl and here packages:
library(tidyverse)
```

```
library(readxl)
library(writexl)
library(here)
```

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(here::here("data", "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(here::here("data", "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 dataset,~ ...2 ...3 ...4 ...5 ...6 ...7 ...8 ...9
##   <chr>          <chr> <chr> <chr> <chr> <chr> <chr> <chr>
## 1 Source: https://portal.ed~ <NA> <NA> <NA> <NA> <NA> <NA> <NA>
## 2 Accessed: 9/28/2019       <NA> <NA> <NA> <NA> <NA> <NA> <NA>
## 3 <NA>                  <NA> <NA> <NA> <NA> <NA> <NA> <NA>
## 4 year                   month date site tran~ quad side subst~ perce~
## 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(here::here("data", "curation/substrate_cover_curated.xlsx", skip = 4,
                                         na = "this"))

substrate_cover

## # A tibble: 23,938 x 9
##   year month date site transect quad side substrate_type percent_cover
##   <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
## 1 <NA>  <NA>  <NA>  carp   1      20    i     b      <NA>
## 2 2000  9     <NA>  carp   1      20    o     b      <NA>
## 3 2000  9     9/8/00 carp   1      20    i     b      100
## 4 2000  9     9/8/00 carp   1      20    o     b      100
## 5 2000  9     9/8/00 carp   1      40    i     b      100
## 6 2000  9     9/8/00 carp   1      40    o     b      100
## 7 2000  9     9/8/00 carp   2      20    i     b      90
## 8 2000  9     9/8/00 carp   2      20    o     b      80
## 9 2000  9     9/8/00 carp   2      40    i     b      80
## 10 2000 9     9/8/00 carp   2      40    o     b      85
## # ... with 23,928 more rows
```

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(here::here("data", "substrate_cover_curated.xlsx", range =
```

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(here::here("data", "substrate_cover_curated.xlsx", range =
```

```
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(here::here("data", "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(here::here("data", "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(here::here("data", "kelp_counts_curated.xlsx"), sheet =
```

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(here::here("data", "kelp_counts_curated.xlsx", sheet = 6))

kelp_golb

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

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 <- here::here("data", "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, here::here("data", "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(here::here("data", "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, here::here("data", "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(here::here("data", "invert_counts_curated.xlsx"), range = "B1:G11")
```

```
# 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

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. Also, if we haven't built a graph with reproducible code, then we might not be able to easily recreate a graph *or* use that code again to make the same style graph with different data.

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

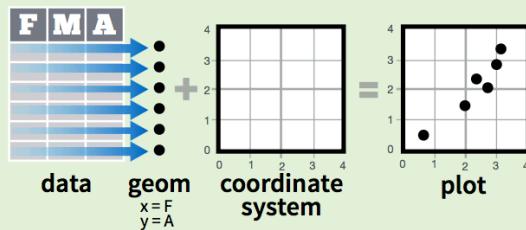
“`ggplot2` implements the grammar of graphics, a coherent system for describing and building graphs. With `ggplot2`, you can do more faster by learning one system and applying it in many places.” - R4DS

So yeah...that `gg` is from “grammar of graphics” - original source

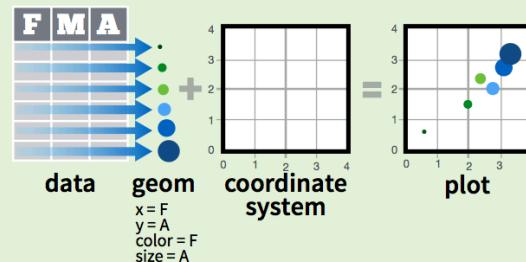
We'll use the `ggplot2` package, but the function we use to initialize a graph will be `ggplot`, which works best for data in tidy format (i.e., a column for every variable, and a row for every observation). Graphics with `ggplot` are built step-by-step, adding new elements as layers with a plus sign (+) between layers (note: this is different from the pipe operator, `%>%`. 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.



6.2 Objectives

- Build several common types of graphs (scatterplot, column, line) in ggplot2
- Customize gg-graph aesthetics (color, style, themes, etc.)
- Update axis labels and titles
- Combine compatible graph types (geoms)
- Build multiseries graphs
- Split up data into faceted graphs
- Exporting figures with `ggsave()`

6.3 Resources

- <https://r4ds.had.co.nz/data-visualisation.html>
- ggplot2-cheatsheet-2.1.pdf
- Graphs with ggplot2 - Cookbook for R
- “Why I use ggplot2” - David Robinson Blog Post

6.4 Lesson

6.4.1 Getting started - Create a new .Rmd, attach packages & get data

Within your existing version-controlled R project, create a new R Markdown document with title “Session 4 - Data visualization with ggplot2.” Remove everything below the first code chunk. Knit and save the .Rmd file within your project working directory as “session_4_ggplot2”.

The `ggplot2` package is part of the `tidyverse`, so we don’t need to attach it separately. Attach the `tidyverse` and `readxl` packages in the top-most code chunk of your .Rmd.

```
library(tidyverse)
library(readxl)
```

In this session, we’ll use data for parks visitation from two files:

- A comma-separated-value (CSV) file containing visitation data for all National Parks in California (`ca_np.csv`)
- A single Excel worksheet containing only visitation for Channel Islands National Park

A bit about Channel Islands National Park: TODO

Add a new code chunk to read in the data from the `data` subfolder within your working directory.

```
ca_np <- read_csv(here::here("data", "ca_np.csv"))
ci_np <- read_xlsx(here::here("data", "ci_np.xlsx"))
```

Let’s take a quick look at the data to see what it contains. For example:

- `View()`: to look at the object in spreadsheet format
- `names()`: to see the variable (column) names
- `summary()`: see a quick summary of each variable

6.4.2 Our first ggplot graph: Visitors to Channel Islands NP

To create a bare-bones ggplot graph, we need to tell R three basic things:

1. We're using `ggplot`
2. Data we're using & variables we're plotting (i.e., what is x and/or y?)
3. What type of graph we're making (the type of `geom`)

Generally, that structure will look like this:

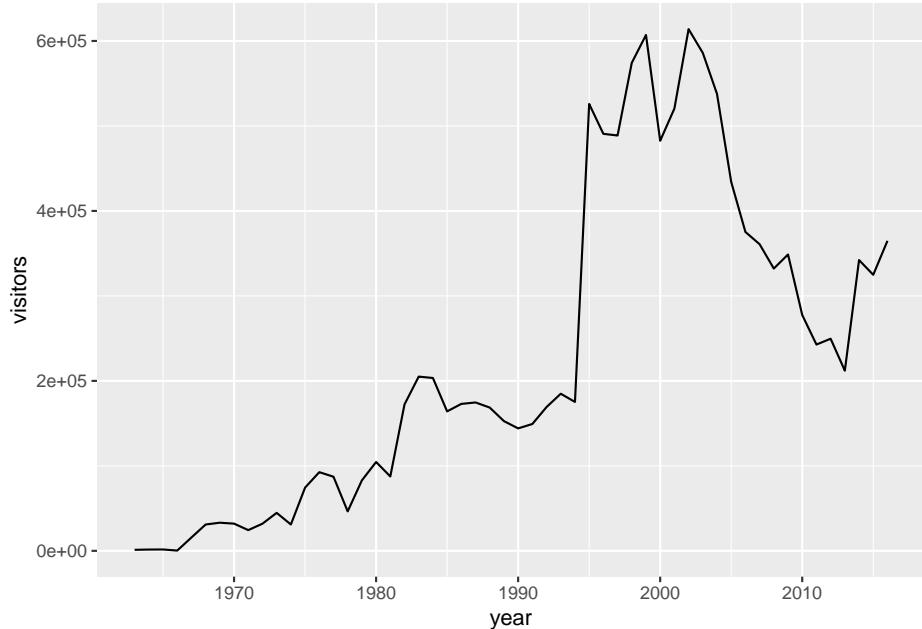
```
ggplot(data = df_name, aes(x = x_var_name, y = y_var_name)) +
  geom_type()
```

Breaking that down:

- First, tell R you're using `ggplot()`
- Then, tell it the object name where variables exist (`data = df_name`)
- Next, tell it the aesthetics `aes()` to specify which variables you want to plot
- Then add a layer for the type of geom (graph type) with `geom_*`() - for example, `geom_point()` is a scatterplot, `geom_line()` is a line graph, `geom_col()` is a column graph, etc.

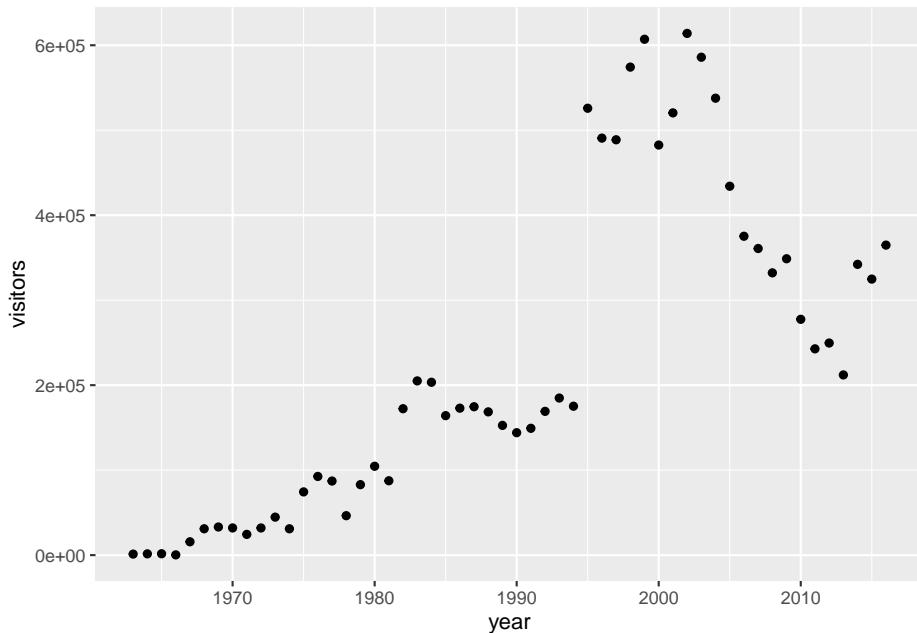
Let's do that to create a line graph of visitors to Channel Islands National Park:

```
ggplot(data = ci_np, aes(x = year, y = visitors)) +
  geom_line()
```



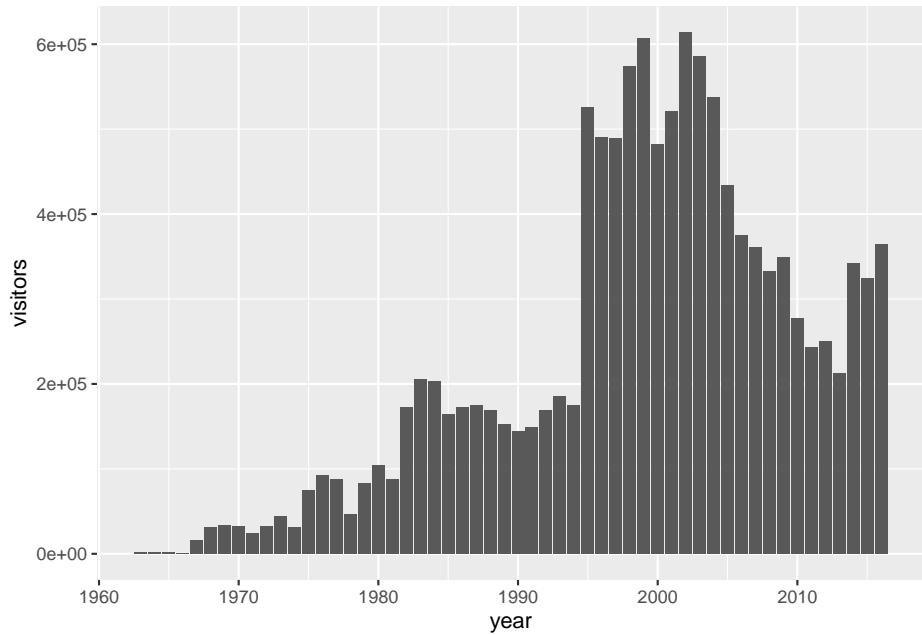
Or, we could change that to a scatterplot just by updating the `geom_*`:

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



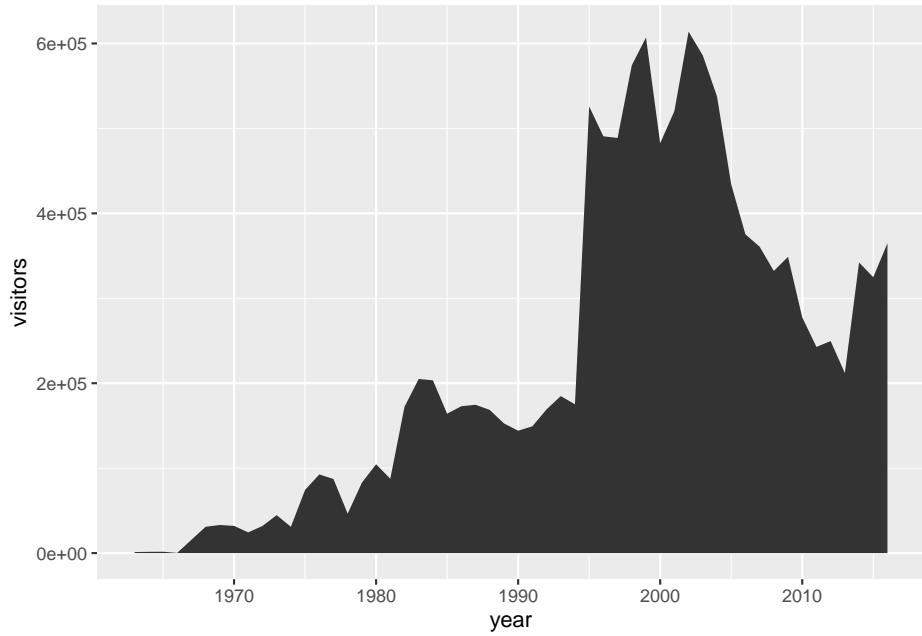
We could even do that for a column graph:

```
ggplot(data = ci_np, aes(x = year, y = visitors)) +  
  geom_col()
```



Or an area plot...

```
ggplot(data = ci_np, aes(x = year, y = visitors)) +
  geom_area()
```



We can see that updating to different `geom_*` types is quick, so long as the types

of graphs we're switching between are compatible.

The data are there, now let's learn some customization.

6.4.3 Intro to customizing ggplot graphs

First, we'll customize some aesthetics (e.g. colors, styles, axis labels, etc.) of our graphs based on non-variable values.

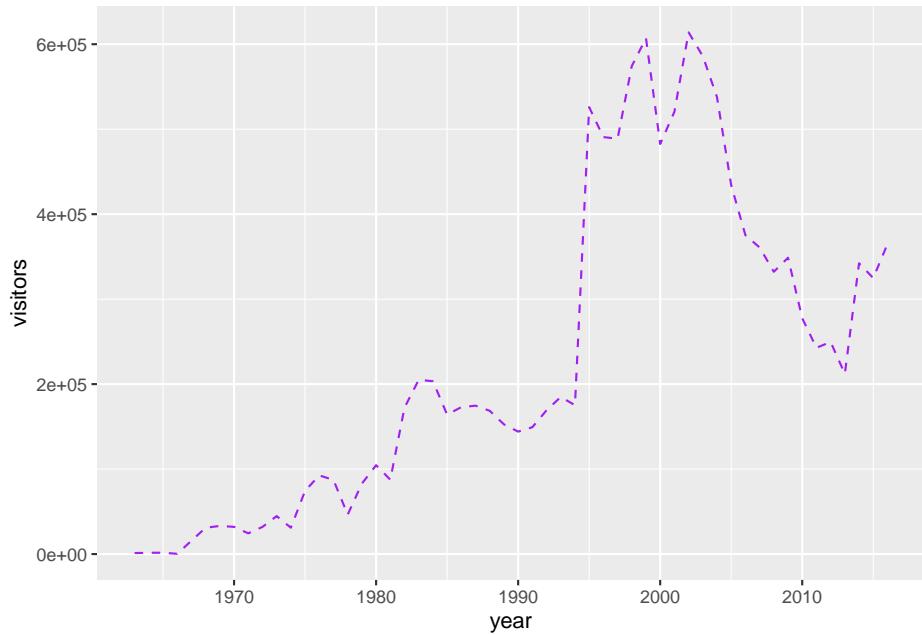
We can change the aesthetics of elements in a ggplot graph by adding arguments within the layer where that element is created.

Some common arguments we'll use first are:

- `color =` or `colour =`: update point or line colors
- `fill =`: update fill color for objects with areas
- `linetype =`: update the line type (dashed, long dash, etc.)
- `pch =`: update the point style
- `size =`: update the element size (e.g. of points or line thickness)
- `alpha =`: update element opacity (1 = opaque, 0 = transparent)

Building on our first line graph, let's update the line color to "purple" and make the line type "dashed":

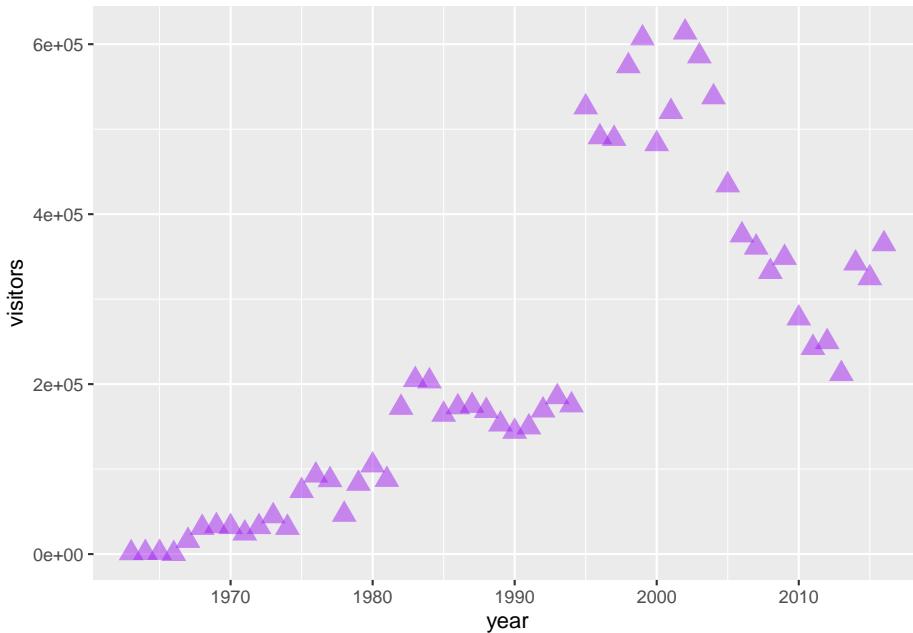
```
ggplot(data = ci_np, aes(x = year, y = visitors)) +
  geom_line(
    color = "purple",
    linetype = "dashed"
  )
```



How do we know which color names ggplot will recognize? If you google “R colors ggplot2” you’ll find a lot of good resources. Here’s one: SAPE ggplot2 colors quick reference guide

Now let’s update the point, style and size of points on our previous scatterplot graph using `color =`, `size =`, and `pch =` (see `?pch` for the different point styles, which can be further customized).

```
ggplot(data = ci_np, aes(x = year, y = visitors)) +
  geom_point(color = "purple",
             pch = 17,
             size = 4,
             alpha = 0.5)
```



In the examples above, we have customized aesthetics based on constants that we input as arguments (e.g., the color / style / size isn't changing based on a variable characteristic or value). Sometimes, however, we **do** want the aesthetics of a graph to depend on a variable.

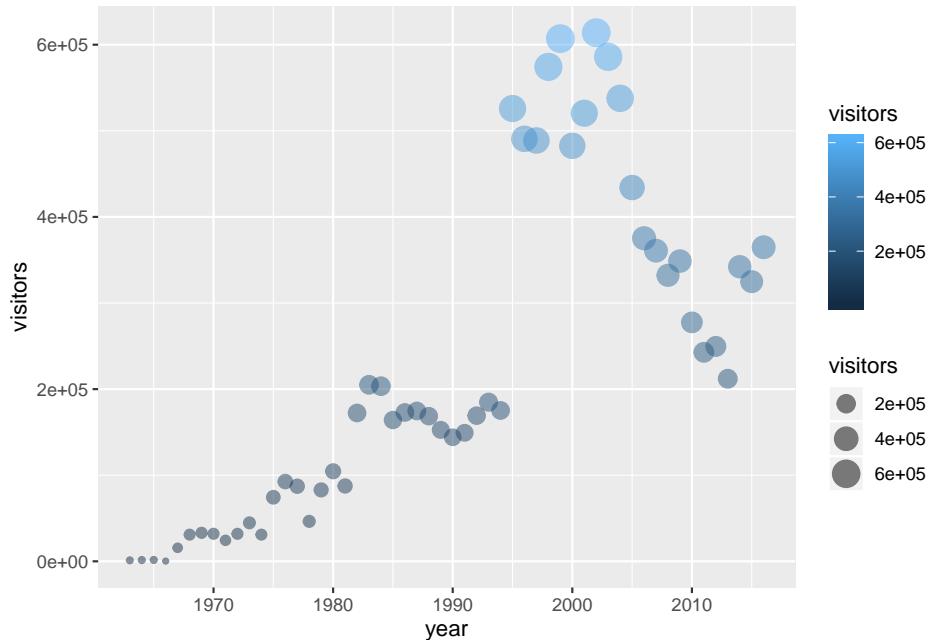
When we want to customize a graph element based on a variable's characteristic or value, add the argument within `aes()` in the appropriate `geom_*`() layer

In short, if updating aesthetics based on a variable, make sure to put that argument inside of `aes()`.

Example: Create a ggplot scatterplot graph where the `size` and `color` of the points change based on the **number of visitors**, and make all points the same level of opacity (`alpha = 0.5`). Notice the `aes()` around the `size =` and `color =` arguments.

Also: this is overmapped and unnecessary. Avoid excessive / overcomplicated aesthetic mapping in data visualization.

```
ggplot(data = ci_np, aes(x = year, y = visitors)) +
  geom_point(
    aes(size = visitors,
        color = visitors),
    alpha = 0.5
  )
```



In the example above, notice that the two arguments that **do** depend on variables are within `aes()`, but since `alpha = 0.5` doesn't depend on a variable then it is *outside the `aes()` but still within the `geom_point()` layer*.

6.4.4 ggplot2 complete themes

While every element of a ggplot graph is manually customizable, there are also built-in themes (`theme_*`()) that you can add to your ggplot code to make some major headway before making smaller tweaks manually.

Here are a few to try today (but also notice all the options that appear as we start typing `theme_` into our ggplot graph code!):

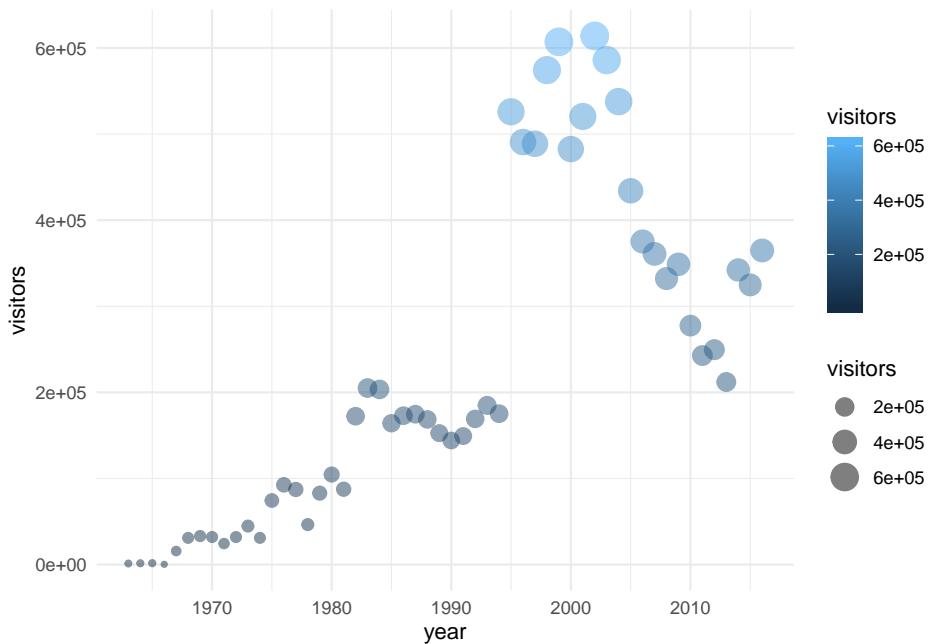
- `theme_light()`
- `theme_minimal()`
- `theme_bw()`

Within those themes, we can also add graph-wide options for fonts and sizes. For example:

- `base_size` := update base font size
- `base_family` := update based font family
- `base_line_size`: update base line size

Here, let's update our previous graph with `theme_minimal()`:

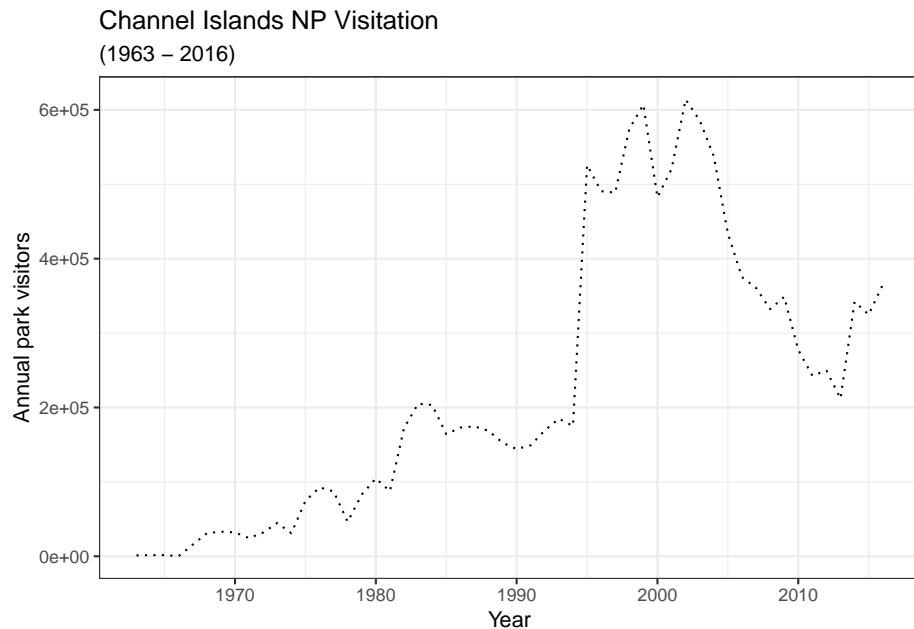
```
ggplot(data = ci_np, aes(x = year, y = visitors)) +
  geom_point(
    aes(size = visitors,
        color = visitors),
    alpha = 0.5
  ) +
  theme_minimal(
  )
```



6.4.5 Updating axis labels and titles

Use `labs()` to update axis labels, and add a title and/or subtitle to your ggplot graph.

```
ggplot(data = ci_np, aes(x = year, y = visitors)) +
  geom_line(linetype = "dotted") +
  theme_bw() +
  labs(
    x = "Year",
    y = "Annual park visitors",
    title = "Channel Islands NP Visitation",
    subtitle = "(1963 - 2016)"
  )
```



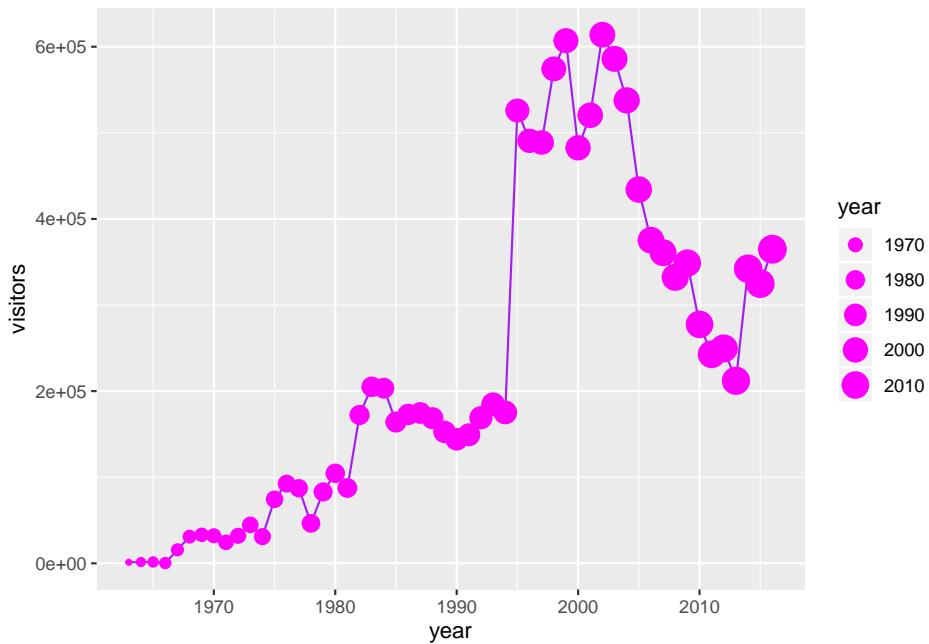
Note: If you want to update the formatting of axis values (for example, to convert to comma format instead of scientific format above), you can use the `scales` package options (see more from the R Cookbook).

6.4.6 Combining compatible geoms

As long as the geoms are compatible, we can layer them on top of one another to further customize a graph.

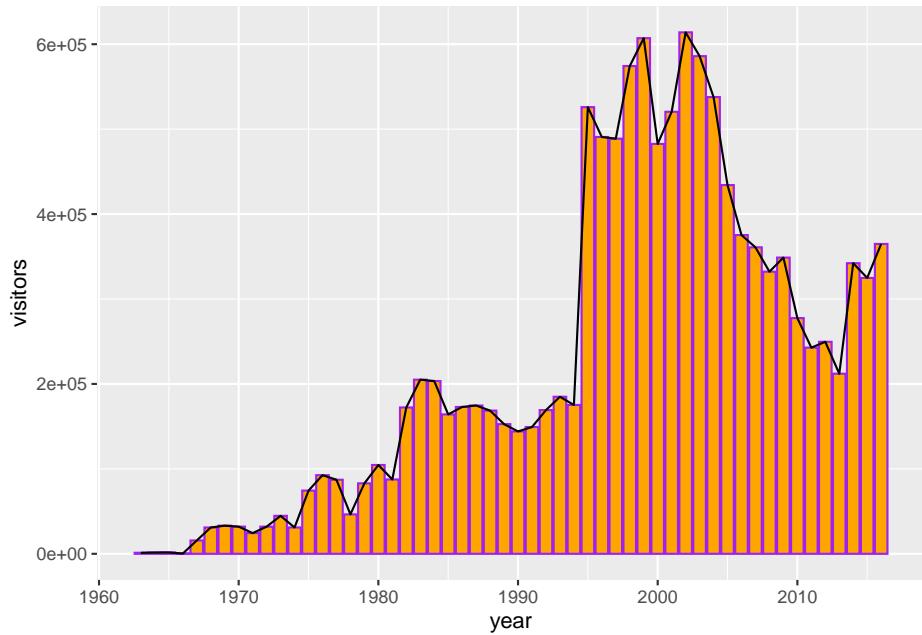
For example, adding points to a line graph:

```
ggplot(data = ci_np, aes(x = year, y = visitors)) +
  geom_line(color = "purple") +
  geom_point(color = "magenta",
             aes(size = year))
```



Or, combine a column and line graph (not sure why you'd want to do this, but you can):

```
ggplot(data = ci_np, aes(x = year, y = visitors)) +
  geom_col(fill = "orange",
           color = "purple") +
  geom_line()
```

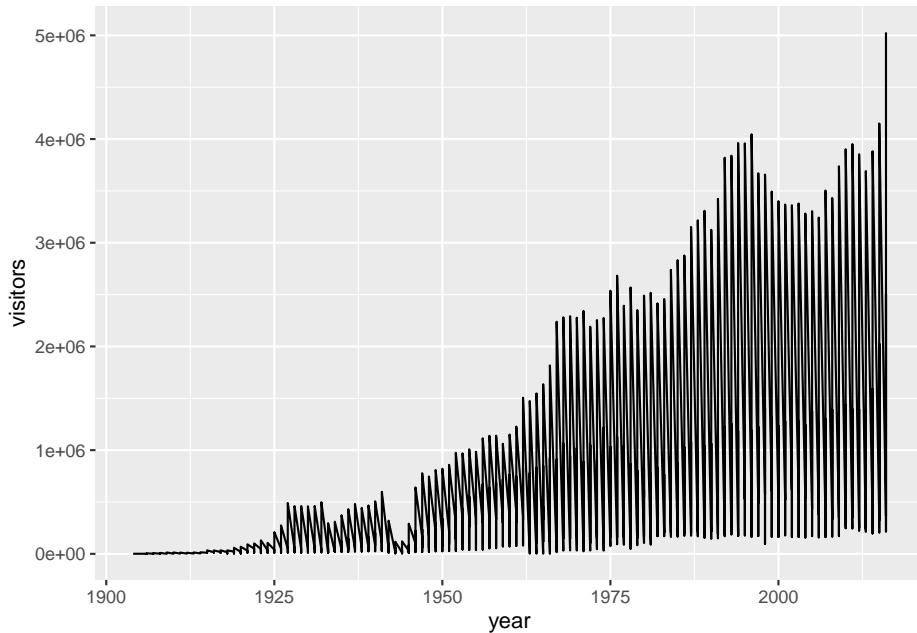


6.4.7 Multi-series ggplot graphs

In the examples above, we only had a single series - visitation at Channel Islands National Park. Often we'll want to visualize multiple series. For example, from the `ca_np` object we have stored, we might want to plot visitation for *all* California National Parks.

To do that, we need to add an aesthetic that lets `ggplot` know how things are going to be grouped. A demonstration of why that's important - what happens if we *don't* let `ggplot` know how to group things?

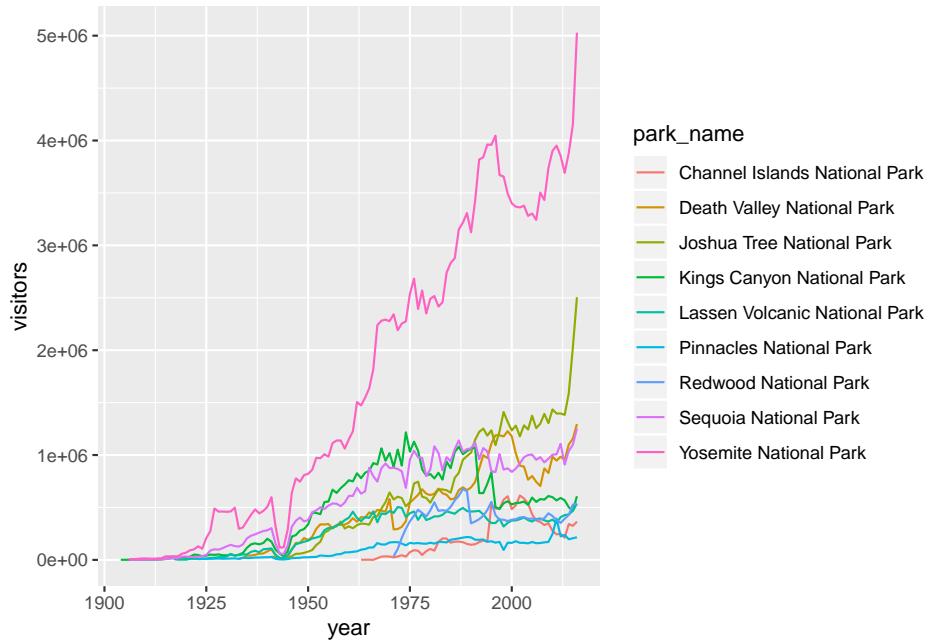
```
ggplot(data = ca_np, aes(x = year, y = visitors)) +
  geom_line()
```



Well that's definitely a mess, and it's because ggplot has no idea that these **should be different series based on the different parks that appear in the 'park_name' column.**

We can make sure R does know by updating an aesthetic based on *park_name*:

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



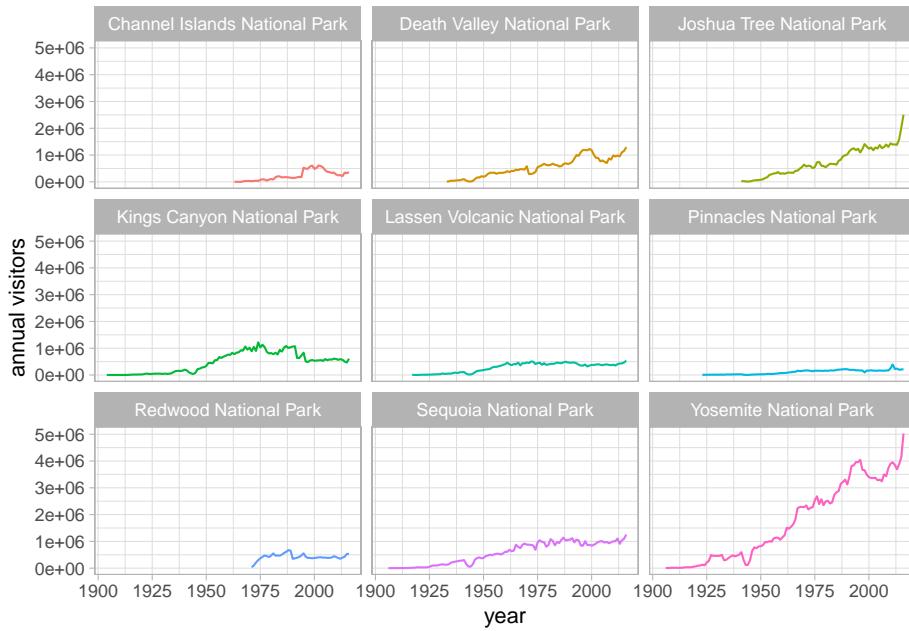
Note: You could also add that aesthetic (`color = park_name`) in the `geom_line()` layer, instead of in the topmost `ggplot()` layer.

6.4.8 Faceting ggplot graphs

When we facet graphs, we split them up into multiple plotting panels, where each panel contains a subset of the data. In our case, we'll split the graph above into different panels, each containing visitation data for a single park.

Also notice that any general theme changes made will be applied to *all* of the graphs.

```
ggplot(data = ca_np, aes(x = year, y = visitors, color = park_name)) +
  geom_line(show.legend = FALSE) +
  theme_light() +
  labs(x = "year", y = "annual visitors") +
  facet_wrap(~ park_name)
```



6.4.9 Exporting a ggplot graph with ggsave()

If we want our graph to appear in a knitted html, then we don't need to do anything else. But often we'll need a saved image file, of specific size and resolution, to share or for publication.

`ggsave()` will export the *most recently run* ggplot graph by default (`plot = last_plot()`), unless you give it the name of a different saved ggplot object. Some common arguments for `ggsave()`:

- `width` := set exported image width (default inches)
- `height` := set exported image height (default height)
- `dpi` := set dpi (dots per inch)

So to export the faceted graph above at 180 dpi, width a width of 8" and a height of 7", we can use:

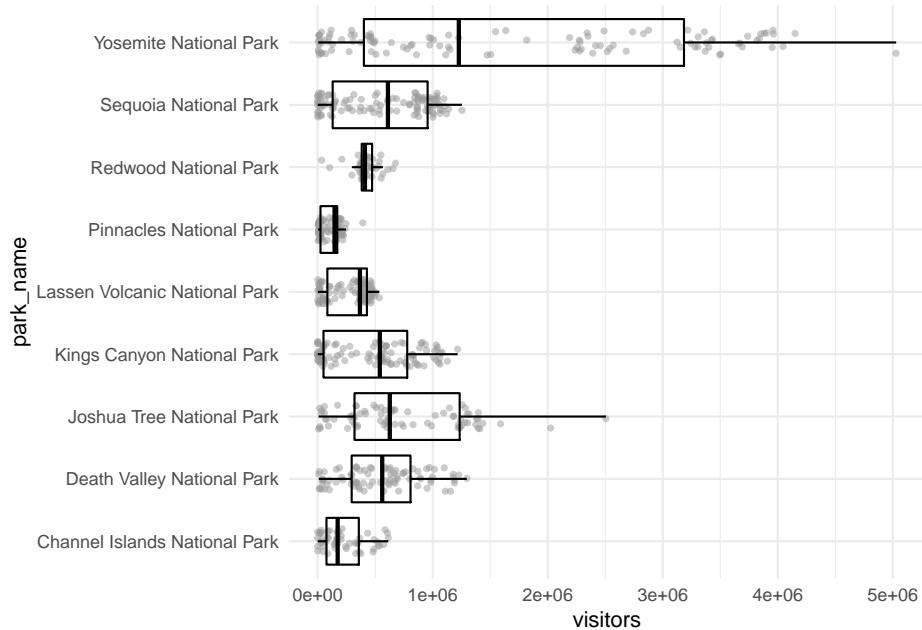
```
ggsave("np_graph.jpg", dpi = 180, width = 8, height = 7)
```

Notice that a .jpg image of that name and size is now stored in your project working directory. You can change the type of exported image, too (e.g. pdf, tiff, eps, png, mmp, svg).

6.4.10 One final graph example: jitter and boxplots

For the record: this is not a good option for showing the visitation data because values are not independent observations of a random variable. But, for the purposes of showing a graph, we'll use visitation as our continuous measured variable in a jitter + boxplot anyway.

```
ggplot(data = ca_np, aes(x = park_name, y = visitors)) +
  geom_jitter(alpha = 0.5,
              color = "gray60",
              width = 0.2,
              size = 1) +
  geom_boxplot(fill = NA,
               color = "black",
               outlier.color = NA) +
  coord_flip() +
  theme_minimal()
```



6.5 Activity 1

6.6 Activity 2

6.7 Efficiency Tips

Chapter 7

Tidying

7.1 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.

7.2 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)

7.3 Resources

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

7.4 Lesson

7.4.1 Lesson Prep

7.4.1.1 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)
```

7.4.1.2 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()`

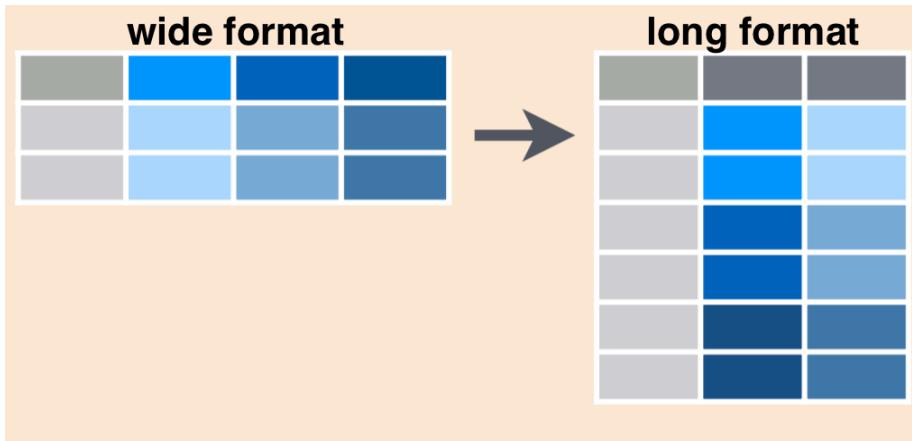
7.4.2 Reshaping with `tidyr::pivot_longer()` and `tidyr::pivot_wider()`

7.4.2.1 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 `tidyverse::pivot_longer()` looks like this:

TODO: Add pivot_longer() schematic

We'll use `tidyverse::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 <- tidyverse::pivot_longer(data = inverts_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  califonia cone snail 2016     451
## 2 7     abur  califonia cone snail 2017      28
## 3 7     abur  califonia cone snail 2018     762
## 4 7     abur  califonia spiny lobster 2016     17
## 5 7     abur  califonia spiny lobster 2017     17
## 6 7     abur  califonia 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 <- inverts_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"
```

7.4.2.2 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...

7.4.2.3 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_dups()`: 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 <- inverts_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`

7.4.3 Combine or separate information in columns with `tidyverse::unite()` and `tidyverse::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

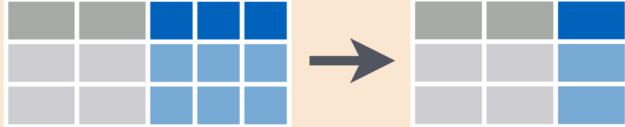
Sebastes miniatus

vermillion

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

7.4.3.1 `tidy::unite()` to merge information from separate columns

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



The diagram illustrates a data transformation process. On the left, there is a wide data frame represented by a grid of colored squares. The grid has 4 columns and 4 rows. The first two columns are light gray, the third column is blue, and the fourth column is dark blue. An arrow points from this wide frame to a long data frame on the right. The long data frame also has 4 columns and 4 rows. The first three columns are light gray, and the fourth column is blue, mirroring the structure of the wide frame's columns.

tidy::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 `tidy::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 %>%
  tidyverse::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.

`tidyverse::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 %>%
  tidyverse::unite(col = "year_site_name",
                  c(year, site, common_name),
                  sep = "-")

head(inverts_triple_unite)

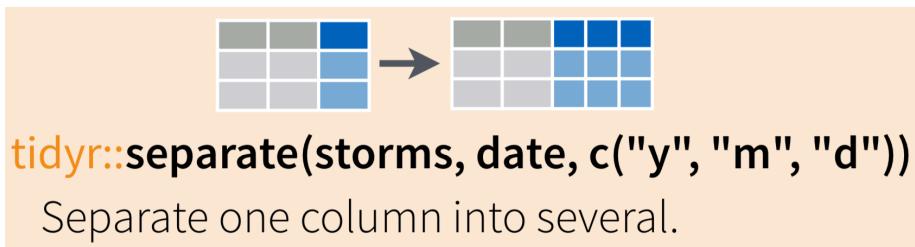
## # A tibble: 6 x 3
##   month year_site_name           sp_count
##   <chr> <chr>                 <dbl>
## 1 7     2016-abur-california cone snail    451
## 2 7     2017-abur-california cone snail    28
## 3 7     2018-abur-california cone snail   762
## 4 7     2016-abur-california spiny lobster  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

7.4.3.2 `tidy::separate()` to separate information into multiple columns

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



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 <- inverts_triple_unite %>%
  tidy::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 <- inverts_triple_unite %>%
  tidyrr::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 `tidyrr::separate()` to override:

```
inverts_sep_all <- inverts_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>
```

```

##   <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.

7.4.4 Convert between explicit and implicit missings (NA)

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 <- inverts_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 <- inverts_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.

7.4.5 Activities

TODO

7.4.6 Fun facts / insights

TODO

Chapter 8

Dplyr and vlookups

8.1 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.

8.2 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

8.3 Resources

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

8.4 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.

8.4.1 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.

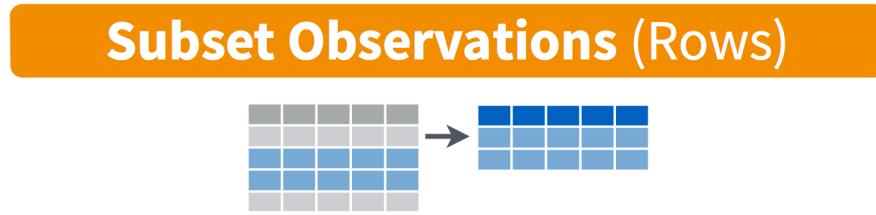


Figure 8.1: `dplyr::filter()`

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()`.

8.4.1.1 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")
```

8.4.1.2 Filter rows based on numeric conditions

Use expected operators ($>$, $<$, \geq , \leq , $=$) 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 =

8.4.1.3 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 == "black surfperch")
```

8.4.1.4 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")
```

8.4.1.5 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
```

8.4.1.6 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

```

8.4.1.7 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

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

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

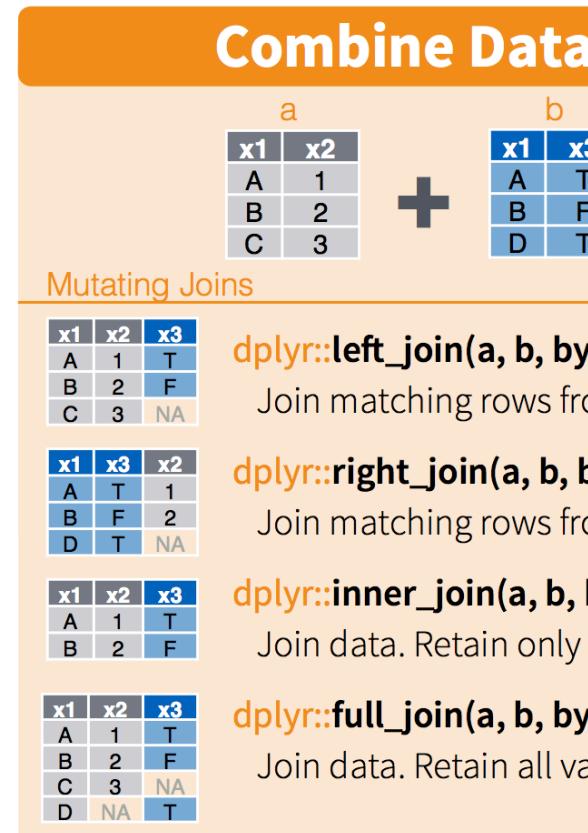
There are a number of ways to merge data frames in R. We’ll use `full_join()`, `left_join()`, and `inner_join()` in this session.

From R Documentation (`?join`):

- `dplyr::full_join()`: “returns all rows and all columns from both x and y. Where there are not matching values, returns NA for the one missing.”

Basically, nothing gets thrown out, even if a match doesn't exist - making `full_join()` the safest option for merging data frames. When in doubt, `full_join()`.

- `dplyr::left_join()`: “return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.”
- `dplyr::inner_join()`: “returns all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned.” This will drop observations that don't have a match between the merged data frames, which makes it a riskier merging option if you're not sure what you're trying to do.



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

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

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

8.4.2.1 `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.

8.4.2.2 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`!

8.4.2.3 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_fronds
##   <dbl> <chr> <chr>          <dbl> <chr>        <dbl>
## 1  2016 abur black surfperch         2 7          307
## 2  2016 abur blacksmith            1 7          307
```

```

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

```

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

8.4.2.4 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
```

8.4.2.5 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)

8.5 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.

8.6 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?)

8.7 Efficiency Tips

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

Chapter 9

Synthesis

9.1 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!

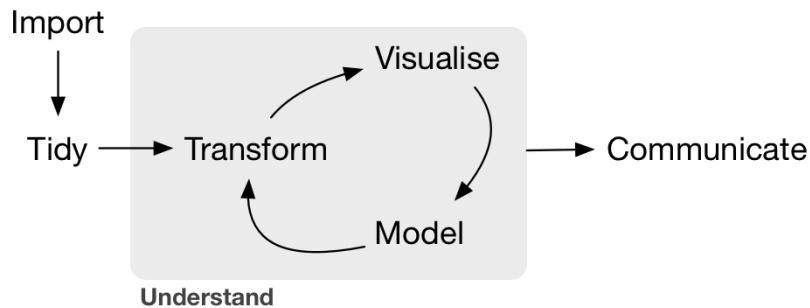


Figure 9.1: Grolemund & Wickham R4DS Illustration

9.2 Objectives

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

- 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

9.3 Resources

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

9.4 Lesson

9.4.1 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`

9.4.2 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.

9.4.3 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()

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

9.6 Interludes (deep thoughts/openescapes)

9.7 Efficiency Tips