

R for Excel Users

Julie Lowndes & Allison Horst

2019-12-19

Contents

1	Welcome	5
1.1	Agenda	5
1.2	Prerequisites	5
2	Overview	7
2.1	Welcome!	7
2.2	Why learn R if I know Excel?	7
2.3	Guiding principles / recurring themes	9
2.4	Resources	10
3	R & RStudio, RMarkdown	11
3.1	Summary	11
3.2	Deep thoughts	12
3.3	Efficiency Tips	12
4	GitHub	13
4.1	Summary	13
4.2	Objectives	13
4.3	Resources	14
4.4	Why should R users use Github?	14
4.5	Setup Git & GitHub	16
4.6	Create a repository on Github.com	18
4.7	Create a gh-pages branch	20
4.8	Clone your repository using RStudio	21
4.9	Inspect your repository	24
4.10	Add files to our local repo	25
4.11	Sync from RStudio to GitHub	27
4.12	Explore remote Github	30
4.13	Create a new R Markdown file	31
4.14	Explore your webpage	32
4.15	Committing - how often? Tracking changes in your files	33
4.16	Happy Git with R	34
4.17	Efficiency Tips	34

5 Graphs with ggplot2	35
5.1 Summary	35
5.2 Lesson	37
6 dplyr and Pivot Tables	55
6.1 Summary	55
6.2 Objectives	55
6.3 Resources	55
6.4 Pivot table overview	56
6.5 RMarkdown setup	56
6.6 Pivot table demo	62
6.7 <code>group_by() %>% summarize()</code>	65
6.8 Oh no, our colleague sent the wrong data!	70
6.9 <code>mutate()</code>	74
6.10 <code>select()</code>	75
6.11 Deep thoughts	76
6.12 Efficiency Tips	76
7 Tidying	77
7.1 Summary	77
7.2 Objectives	77
7.3 Resources	78
7.4 Lesson	78
8 Dplyr and vlookups	91
8.1 Summary	91
8.2 Lessons	92
8.3 Fun / kind of scary facts	104
8.4 Interludes (deep thoughts/openscapes)	104
8.5 Efficiency Tips	104
9 readxl	105
9.1 Summary	105
9.2 Lesson	105
9.3 Efficiency Tips	105
9.4 Additional thoughts	105
10 Synthesis	107
10.1 Summary	107
10.2 Objectives	107
10.3 Resources	108
10.4 Lesson	108
10.5 Fun facts (quirky things) - making a note of these wherever possible for interest (little “Did you know?” sections)	110
10.6 Interludes (deep thoughts/openscapes)	110
10.7 Efficiency Tips	110

Chapter 1

Welcome

Hello! This is a workshop taught by Julie Stewart Lowndes and Allison Horst at the RStudio Conference: January 27-28 in San Francisco, California.

We are environmental scientists who use and teach R in our daily work. We both work at the University of California Santa Barbara: Julie is based at the National Center for Ecological Analysis and Synthesis as part of the Ocean Health Index team and leads Openscapes, and Allison is based at the Bren School of Environmental Science and Management as a lecturer of data science & statistics — and is also an Artist in Residence at RStudio.

1.1 Agenda

Time	Day 1	Day 2
9-10:30 break	Motivation, R & RStudio, RMarkdown (JL)	Tidying data (AH)
11-12:30 lunch	Intro to GitHub (JL)	dplyr & VLOOKUPs (AH)
13:30-15:00 break	ggplot2 & Charts (AH)	Collaborating in #rstudio (JL)
15:30-17:00	dplyr & Pivot Tables (JL)	Synthesis (AH)

1.2 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. Create a an account:
 - <https://github.com>
1. Get comfortable: if you're not in a physical workshop, be set up with two screens if possible. You will be following along in RStudio on your own computer while also watching a virtual training or following this tutorial on your own.

Chapter 2

Overview

TODO: add Star Wars illustrations & Wickham R4DS illustration (as slides?)

2.1 Welcome!

In this workshop you will learn hands-on how to begin to interoperate between Excel and R. But this workshop is not only about learning R; we will learn R using additional software: RStudio and GitHub. These tools will help us develop good habits for working in a reproducible and collaborative way — critical attributes of the modern analyst.

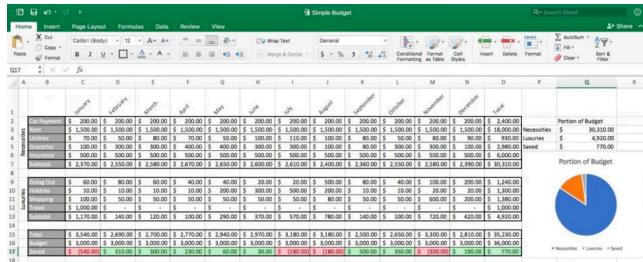
It's going to be fun and empowering!

2.2 Why learn R if I know Excel?

Excel is a widely used and powerful tool for working with data, and it is great for a lot of things. This is convenient and familiar; most of us have had their first experiences with data through Excel or other spreadsheet programs. As Jenny Bryan has said, “Excel is how we learn that we love data analysis”.

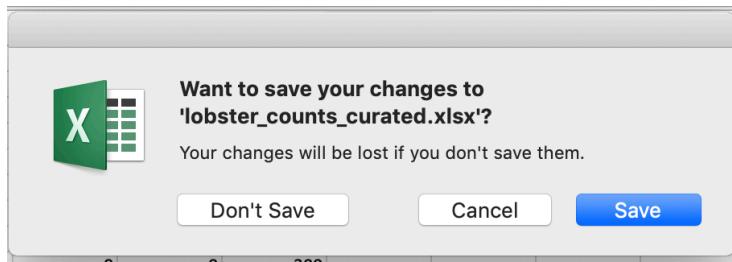
Excel is great for data entry. Can also be good for looking at data and feeling like you can touch it, and creating quick exploratory figures.

Excel can also become problematic with extending to analyses. This is because there aren't firm lines between what is data and what is analyses. For example, in this sheet:



This makes the analytical steps taken are not readily apparent, nor easy to reproduce. 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.

This also makes them pretty brittle/sensitive to minor changes. Has seeing this ever given you a feeling of horror:



So while it is great how easily you can update different fields and add analytical steps in an Excel sheet, it can also be a bit hard to handle, particularly as projects get more complicated.

So, 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.

2.2.1 What to expect

This is going to be a fun workshop.

This workshop will give you hands-on experience and confidence with R, and how to interoperate between Excel and R — it is not about wholesale replacing everything you do in Excel into R. We will learn technical skills that you can incrementally incorporate into your existing workflows. But a big part of interfacing between Excel and R is not only skillsets, it is mindsets. It is the mindset about how we think about data. How we shape data and organize data and analyze data. And how what we do now can make our analytical life better in the future.

A modern R user has a workflow framed around collaboration, and uses an ecosystem of tools and practices. We will be learning three main things all at the same time:

1. coding with best practices (R/RStudio/tidyverse)
2. collaborative bookkeeping (Git/GitHub)
3. reporting and publishing (RMarkdown/GitHub)

R users keep raw data separate from their analyses, which means having data in one file and written computational commands saved as a separate file. We also embrace the concept of “**tidy data**”, where the data has a rectangular shape and each column is a variable and each row is an observation. Tidy data is a way of life.

region	state	code	park_name	type	visitors	year
PW	CA	CHIS	Channel Islands National Park	National Park	1200	1963
PW	CA	CHIS	Channel Islands National Park	National Park	1500	1964
PW	CA	CHIS	Channel Islands National Park	National Park	1600	1965
PW	CA	CHIS	Channel Islands National Park	National Park	300	1966
PW	CA	CHIS	Channel Islands National Park	National Park	15700	1967
PW	CA	CHIS	Channel Islands National Park	National Park	31000	1968
PW	CA	CHIS	Channel Islands National Park	National Park	33100	1969
PW	CA	CHIS	Channel Islands National Park	National Park	32000	1970

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.

You are all welcome here, please be respectful of one another. 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.

2.3 Guiding principles / recurring themes

“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.

Scripted analyses — We write analytical logic in code (rather than clicks) so that can be understood, rerun, and built upon.

Learn from data that are not your own — We aren't using your data in this workshop, but you will see similarities and patterns, and you'll see that these tools and practices apply to your work.

Think ahead for Future You, Future Us. Help make lives easier — first and foremost your own. Create breadcrumbs for yourselves and others: document and share your work.

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
- more to come!

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.

3.1.1 Objectives

In this lesson we will:

- get oriented to the RStudio interface
- explore RMarkdown. This will be your notes.
- discuss RMarkdown files vs Console (running vs knitting)
- learn a few base R functions (`c()`)
- error messages and help pages
- discuss and install packages (`here()`)
- intro pipe operator (`%>%`)
- configure Git (to prepare for next session)

3.1.2 Resources

- R for Excel Users by Gordon Shotwell (blog)

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

3.1.3 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.

3.1.4 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.

3.1.5 What is RMarkdown? (1-minute video)

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

What is RMarkdown?

3.2 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.

3.3 Efficiency Tips

—>

Chapter 4

GitHub

TODO: no github folder, new screenshots, earlier emphasis on syncing steps

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 new R users 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.

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 new R user getting started.

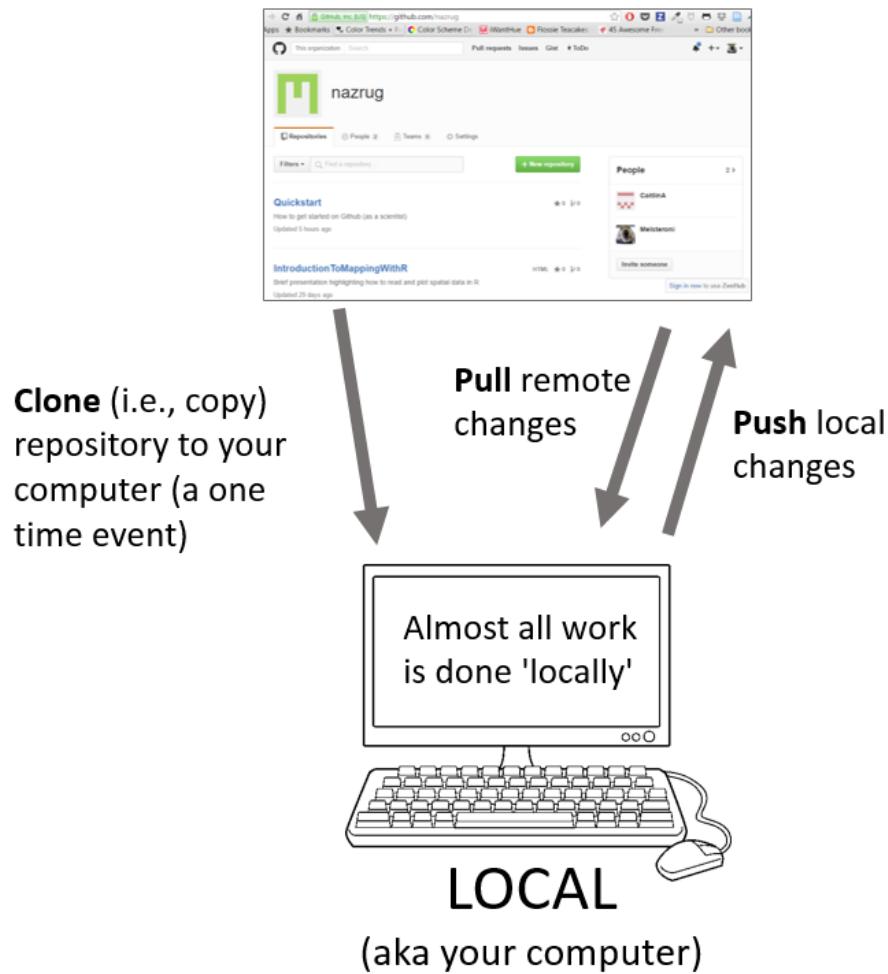
These concepts are more important for coders who want the entire coding community (and not just people working on the same project) to be able to suggest changes to their code. This isn't how most new R users will use Github.

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 users (e.g., datacar-pentry).
- **Repository:** A folder within the organization that includes files dedicated to a project.
- **Local Github:** Copies of Github files located on 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

REMOTE (aka Github website)



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. We 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 = "jules32", user.email = "jules32@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.

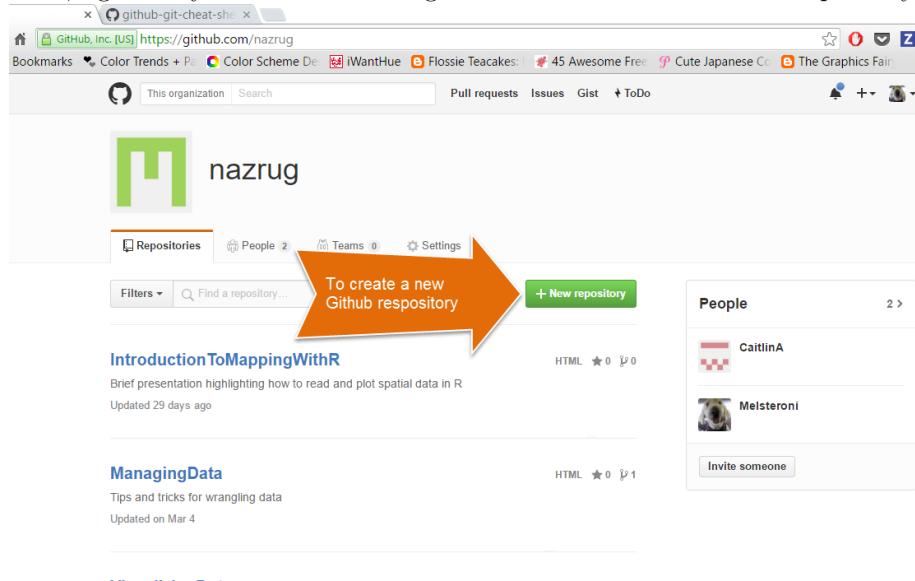
Quit RStudio.

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!

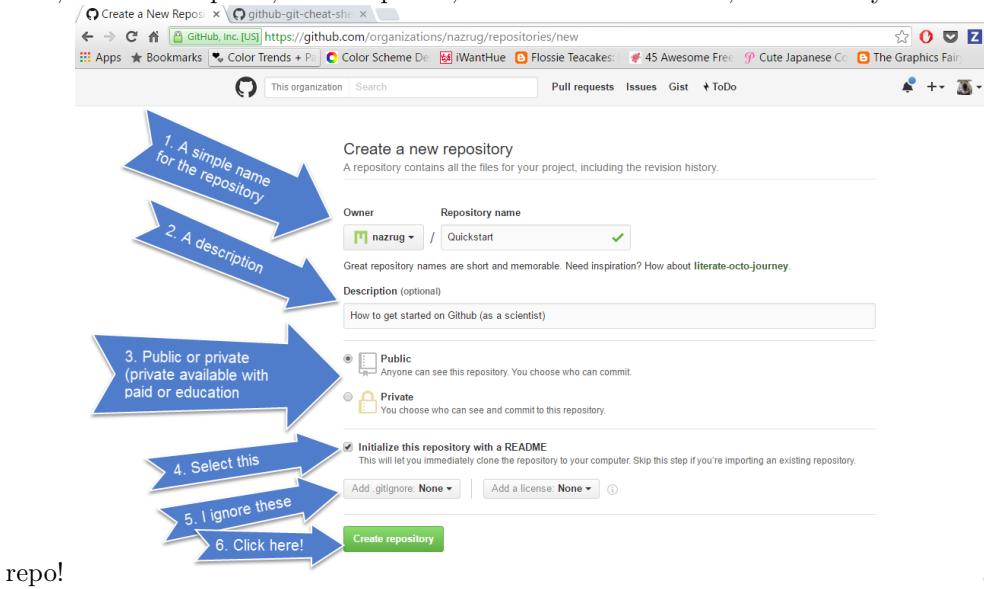
4.6 Create a repository on Github.com

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 **r-workshop**.

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

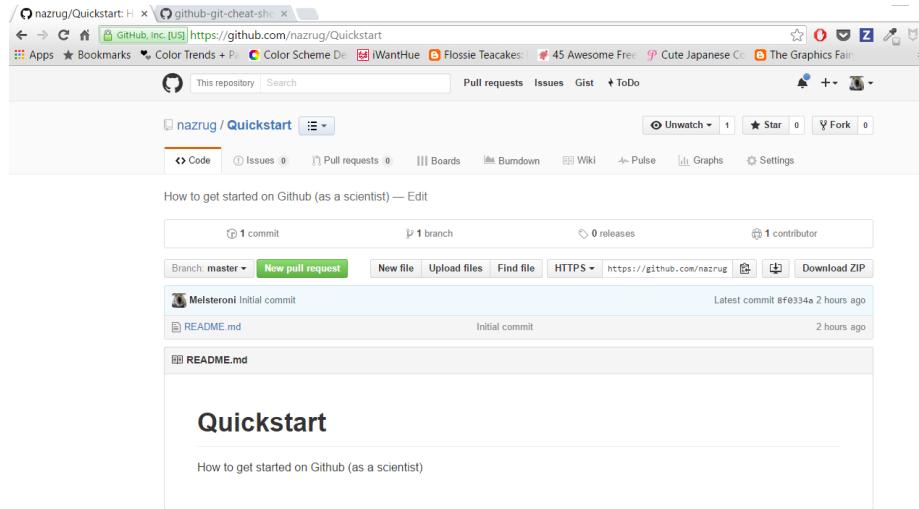


The *Add gitignore* option adds a document where you can identify files or file-types you want Github to ignore. These files will stay in on the local Github folder (the one on your computer), but will not be uploaded onto the web version of Github.

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.



4.7 Create a gh-pages branch

We aren't going to talk about branches very much, but they are a powerful feature of git/GitHub. I think of it as creating a copy of your work that becomes a parallel universe that you can modify safely because it's not affecting your original work. And then you can choose to merge the universes back together if and when you want. By default, when you create a new repo you begin with one branch, and it is named `master`. When you create new branches, you can name them whatever you want. However, if you name one `gh-pages` (all lowercase, with a - and no spaces), this will let you create a website. And that's our plan. So, let's do this to create a `gh-pages` branch:

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

Let's type `gh-pages`.

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

First click to control branches:

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

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

4.8 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).

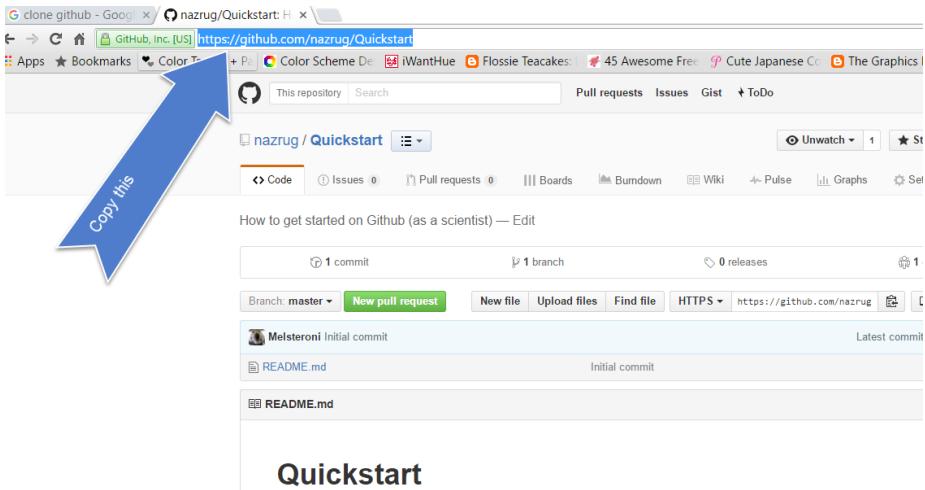
Let's all make a folder called `github` (all lowercase!) in our home directories. So it will look like this:

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

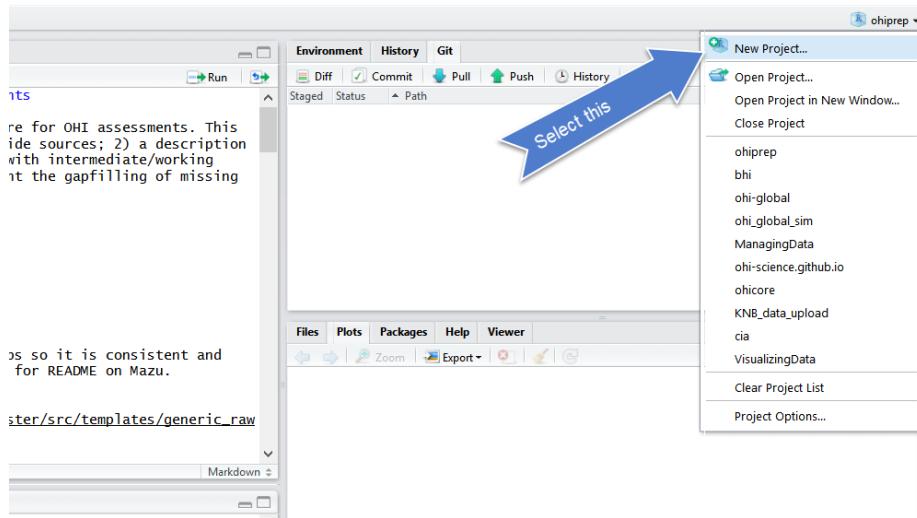
This will let us take advantage of something that is really key about GitHub.com: you can easily navigate through folders within repositories and the urls reflect this navigation. The greatness of this will be evident soon. So let's set ourselves up for easily translating (and remembering) those navigation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

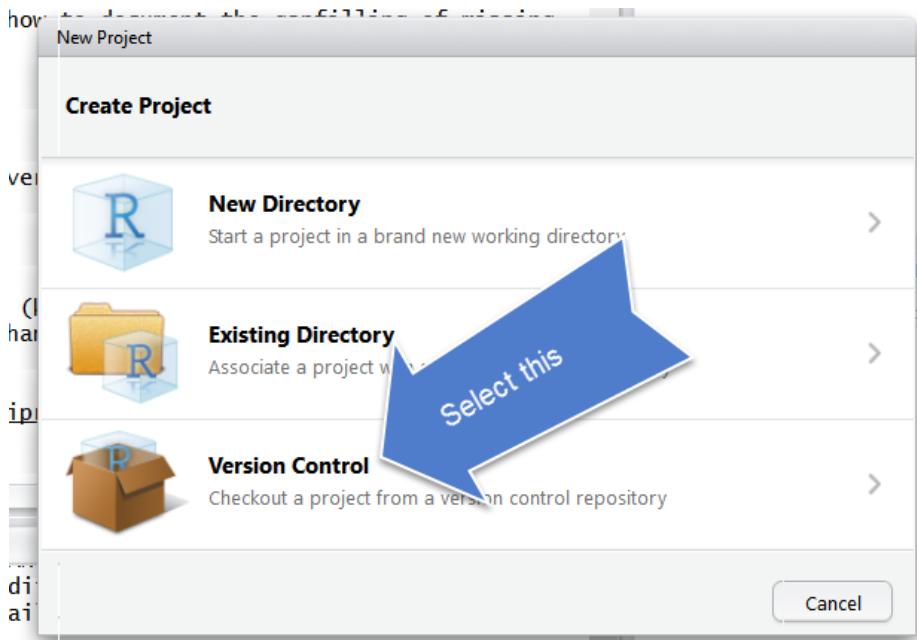
Step 1: Copy the web address of the repository you want to clone.



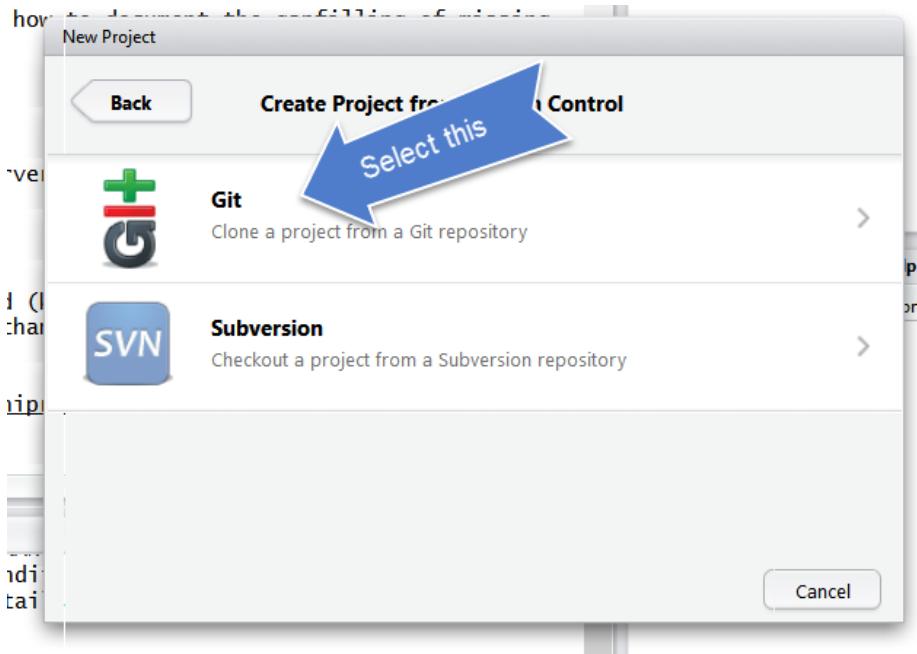
Step 2: from RStudio, go to New Project (also in the File menu).



Step 3: Select Version Control

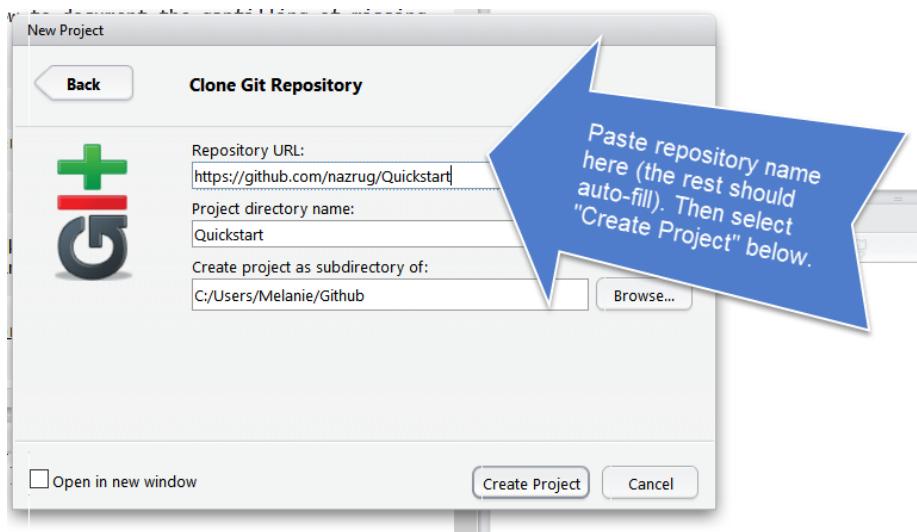


Step 4: Select Git

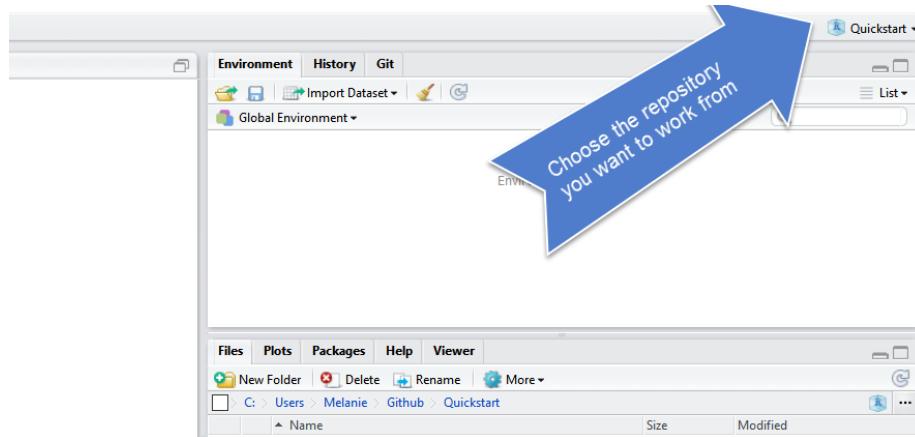


Step 5: Paste it in the Repository URL field, and type **tab** to autofill the Project Directory name. Make sure you keep the Project Directory Name **THE SAME** as the repository name from the URL.

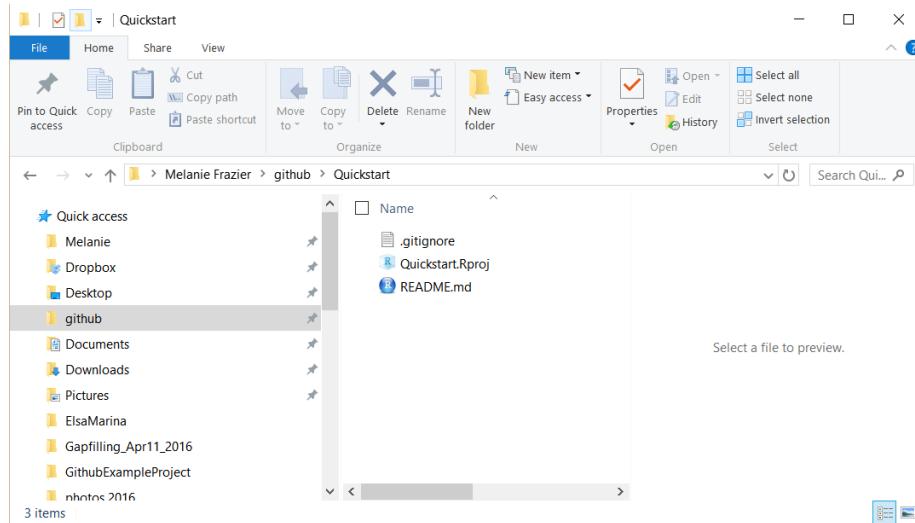
Save it in your github folder (click on Browse) to do this.



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:



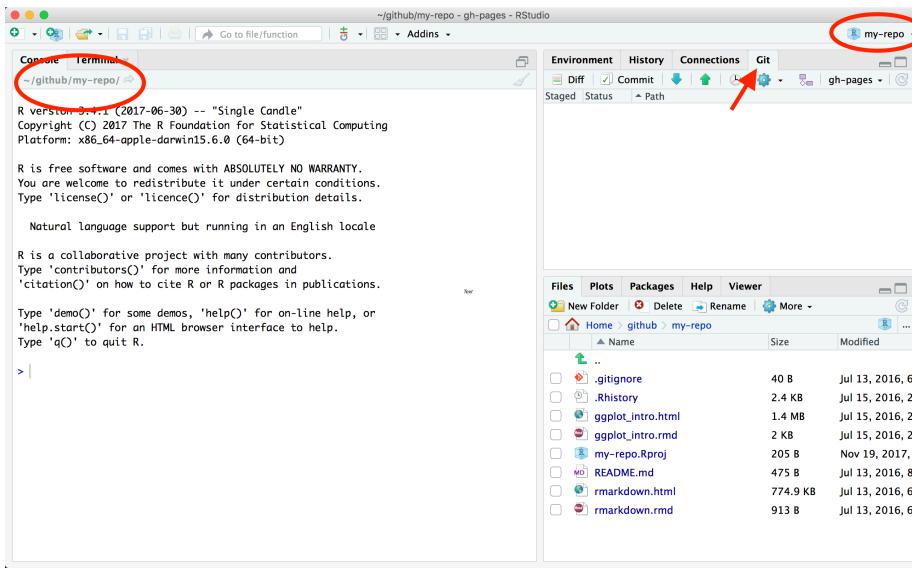
Ta da!!!! The folder doesn't contain much of interest, but we are going to change that.

4.9 Inspect your repository

Notice a few things in our repo here:

1. Our working directory is set to `~/github/r-workshop`. 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.

4.10 Add files to our local repo

The repository will contain:

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

Let's create the following:

- folder called "data"
- folder called "figures"

They both show up in your Finder! ...

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

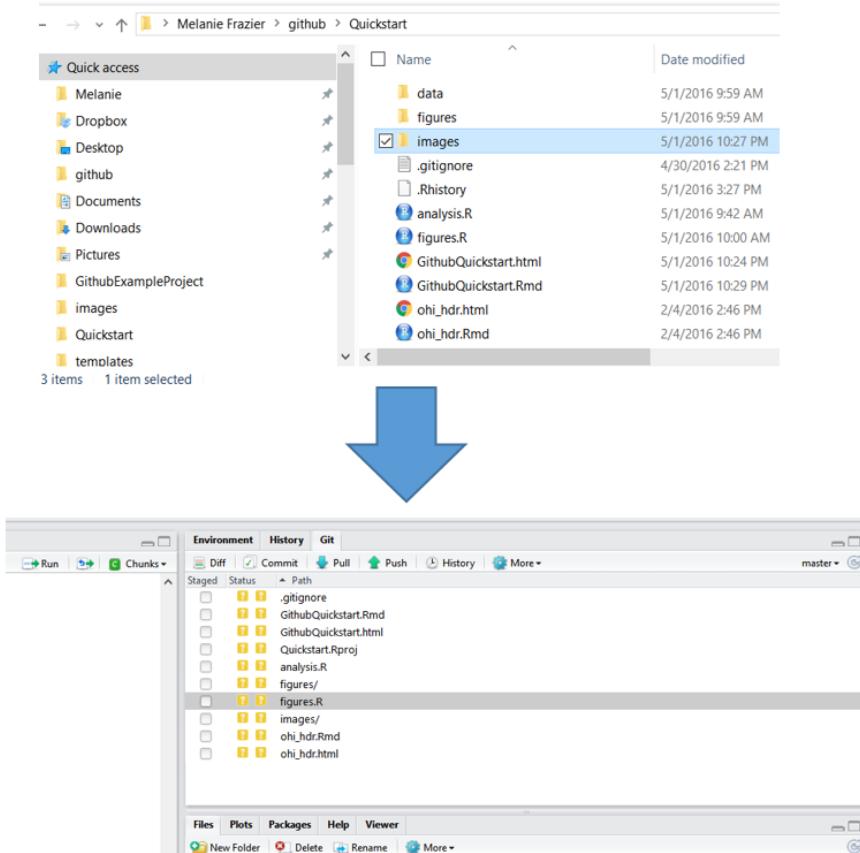
- fish_counts_curated.csv
- invert_counts_curated.xlsx
- kelp_counts_curated.xlsx
- substrate_cover_curated.xlsx
- lobsters.xlsx
- lobsters2.xlsx
- ca_np.csv
- ci_np.xlsx

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.

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).

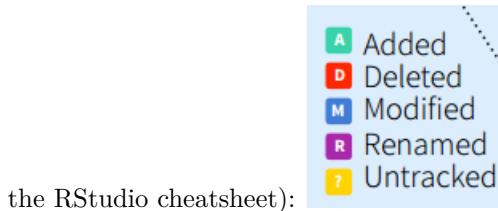
To make changes to the repository, you will work from your computer (“local Github”).

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



4.10.2 Inspect what has changed

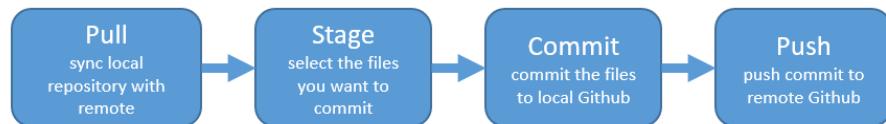
These are the codes RStudio uses to describe how the files are changed, (from



the RStudio cheatsheet):

4.11 Sync from RStudio to GitHub

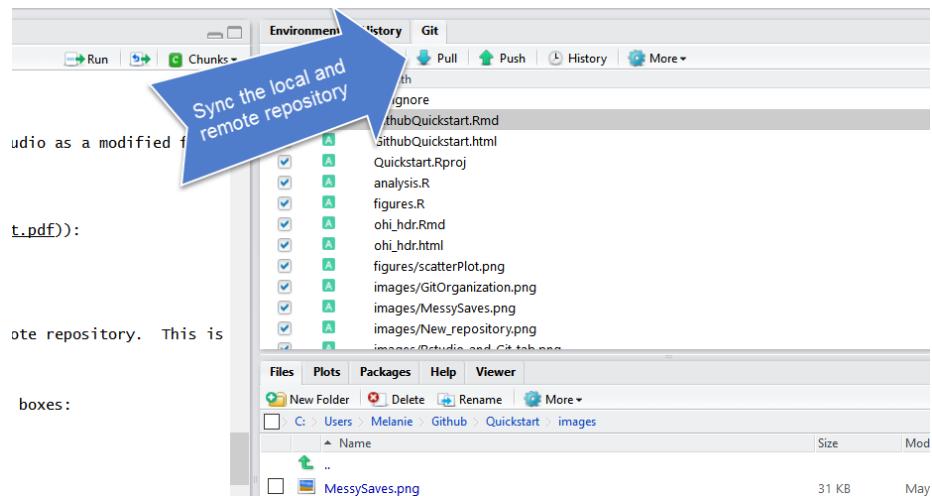
When you are ready to commit your changes, you follow these steps:



We walk through this process below:

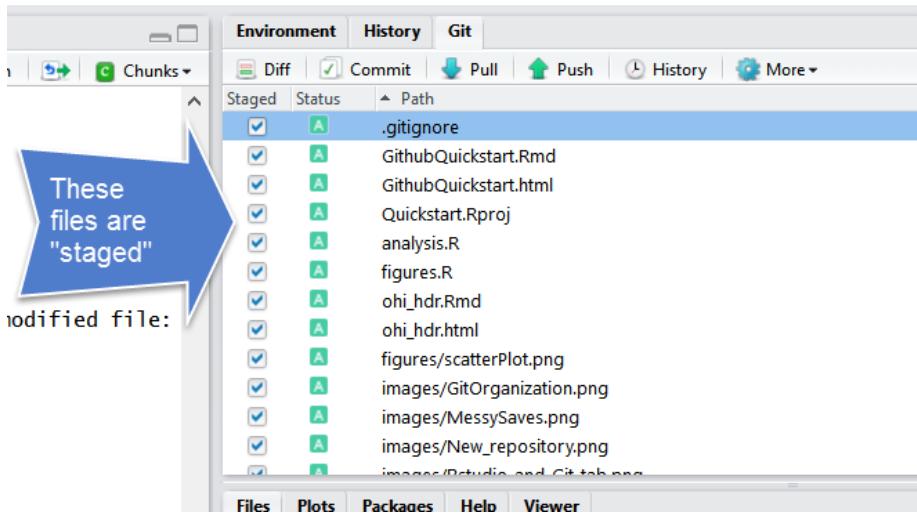
4.11.1 Pull

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.

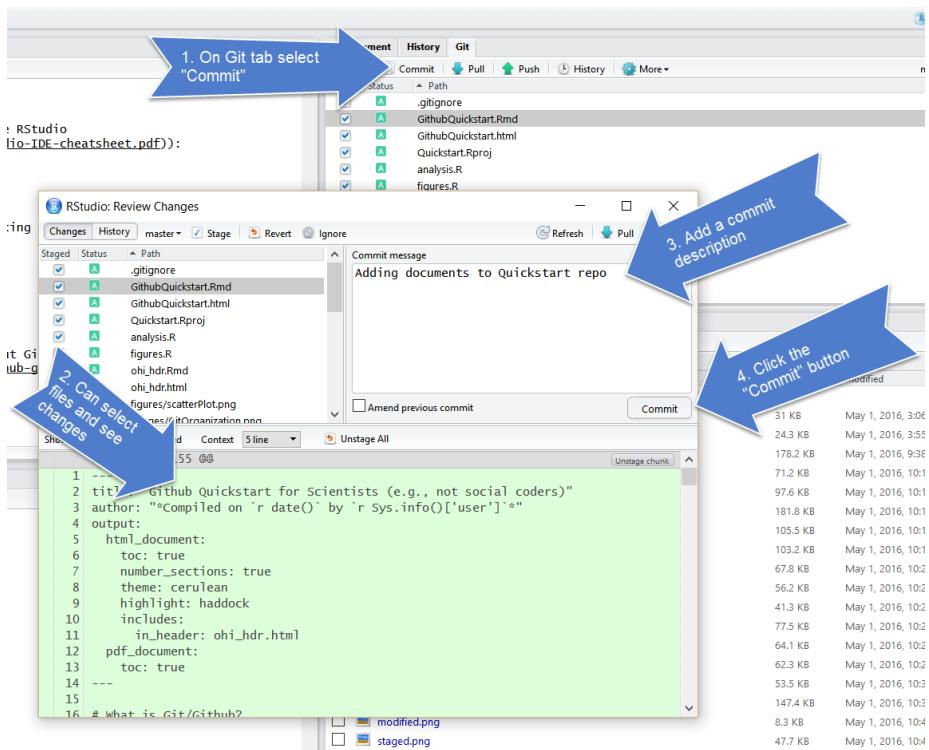


4.11.2 Stage

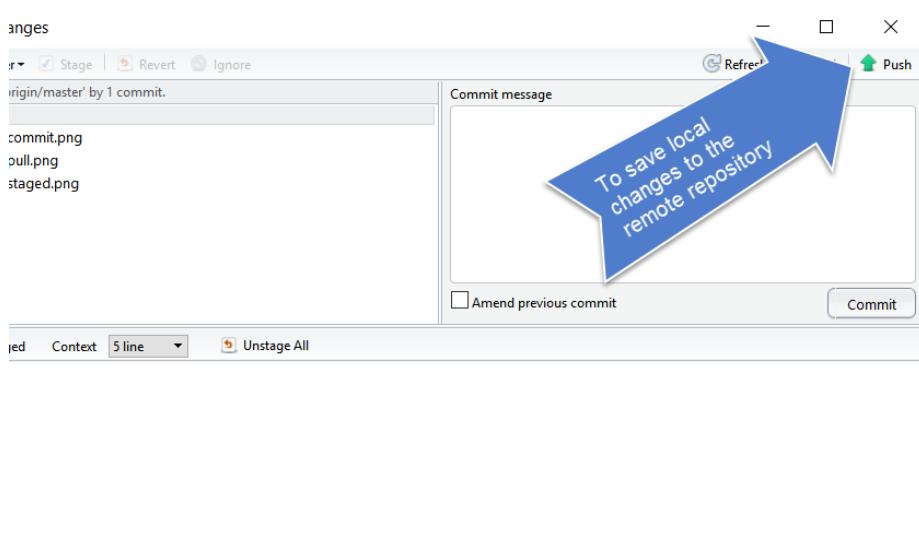
Stage the files you want to commit. In RStudio, this involves checking the “Staged” boxes:



4.11.3 Commit

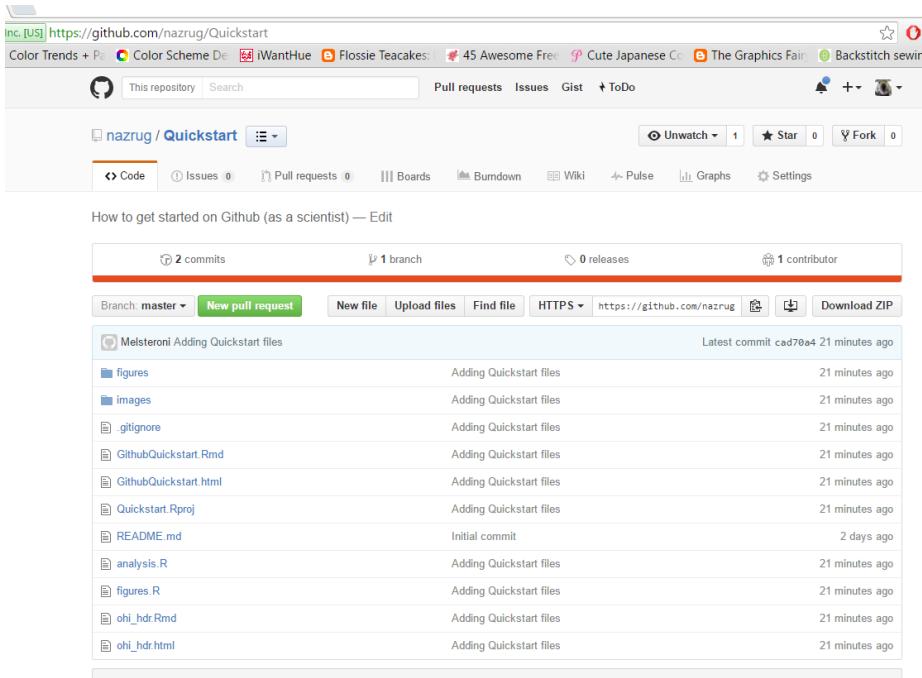


4.11.4 Push



4.12 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 browser extensions and tabs. 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', and 'ToDo'. On the right, there are buttons for 'Unwatch', 'Star', 'Fork', and settings. The main content area shows a message 'How to get started on Github (as a scientist) — Edit'. Below this, there are summary statistics: 2 commits, 1 branch, 0 releases, and 1 contributor. A 'New pull request' button is highlighted in green. The file list shows a commit by 'Melsteroni' adding 'Quickstart files'. The commit details are as follows:

File	Description	Time 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.

4.12.1 Activity

Go back to RStudio.

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 (like your dog's name), save, and see what happens in your Git Tab. Sync it to your remote repository at Github.com.

4.13 Create a new R Markdown file

Now 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, knit your file, and sync your file to GitHub: commit and pull

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 asterisk, there have been changes since it was saved. Remember to save before syncing to GitHub!

4.14 Explore your webpage

You've just created a webpage!

It will exist at this url: `username.github.io/repo-name/filename`. Mine is: `jules32.github.io/r-workshop/readxl`.

Pro Tip Pay attention to URLs. An unsung skill of the modern analyst is to be able to navigate the internet by keeping an eye on patterns.

Troubleshooting:

- 404 error? Remove trailing / from the url
- Wants you to download? Remove trailing .Rmd from the url

4.15 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:

<https://github.com/OHI-Science/ohiprep>

This repository | Search | Pull requests | Issues | Gist | + ToDo

OHI-Science / ohiprep | Unwatch | 20 | ★ Star | 4 | Fork | 2

Code | Issues | Pull requests | Boards | Burndown | Wiki | Pulse | Graphs | Settings

mean 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-Science/ohiprep | Download ZIP

Author	Message	Time Ago
Antarctica	Updating ICOs 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
src	Create README.md	4 days ago

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 diff shows 88 changed files with 2,432,873 additions and 426,265 deletions. A message at the top of the diff area says 'Sorry, we could not display the entire diff because it was too big.' Below this, the diff itself is displayed with lines numbered 22 through 38. Lines 22-26 are in red, indicating deleted code. Lines 34-36 are in green, indicating added code. 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_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_git <- file.path('~/github/ohiprep', goal)

```

4.16 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!

4.17 Efficiency Tips

Chapter 5

Graphs with ggplot2

5.1 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.

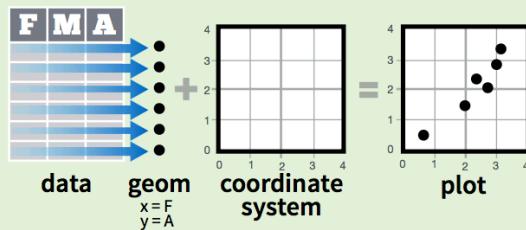
“`ggplot2` implements the grammar of graphics, a coherent system for describing and building graphs. With `ggplot2`, you can do more faster by learning one system and applying it in many places.” - R4DS

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

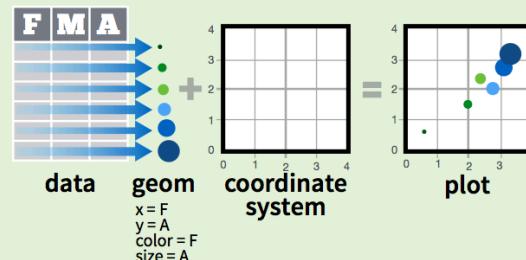
We'll use the `ggplot2` package, but the function we use to initialize a graph will be `ggplot`, which works best for data in tidy format (i.e., a column for every variable, and a row for every observation). Graphics with `ggplot` are built step-by-step, adding new elements as layers with a plus sign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



5.1.1 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()`

5.1.2 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

5.2 Lesson

5.2.1 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 “Data visualization with ggplot2.” Remove everything below the first code chunk. Knit and save the .Rmd file within your project working directory as “my_ggplot2”.

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

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

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 (`ci_np.xlsx`)

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

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

5.2.2 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:

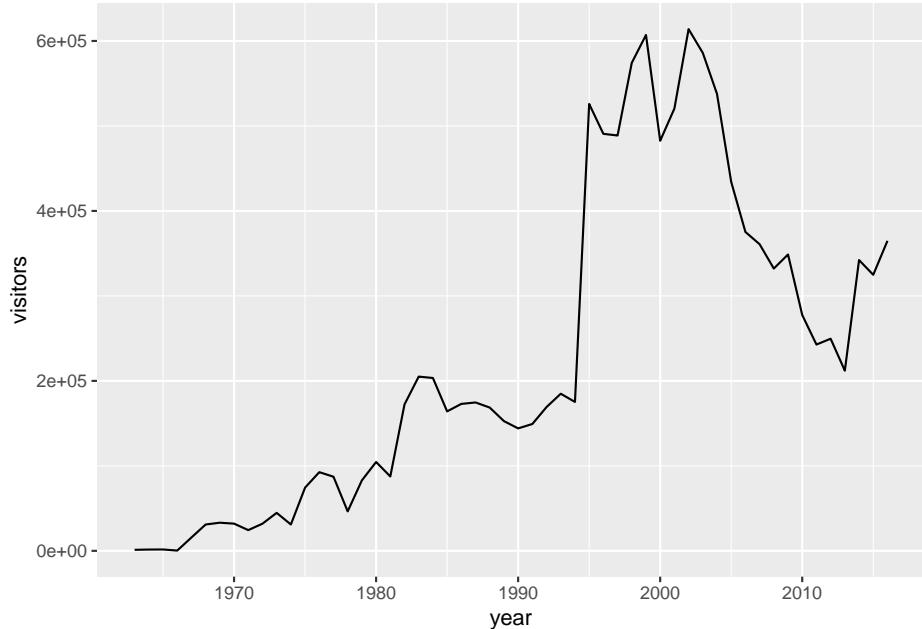
```
ggplot(data = df_name, aes(x = x_var_name, y = y_var_name)) +
  geom_type()
```

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.

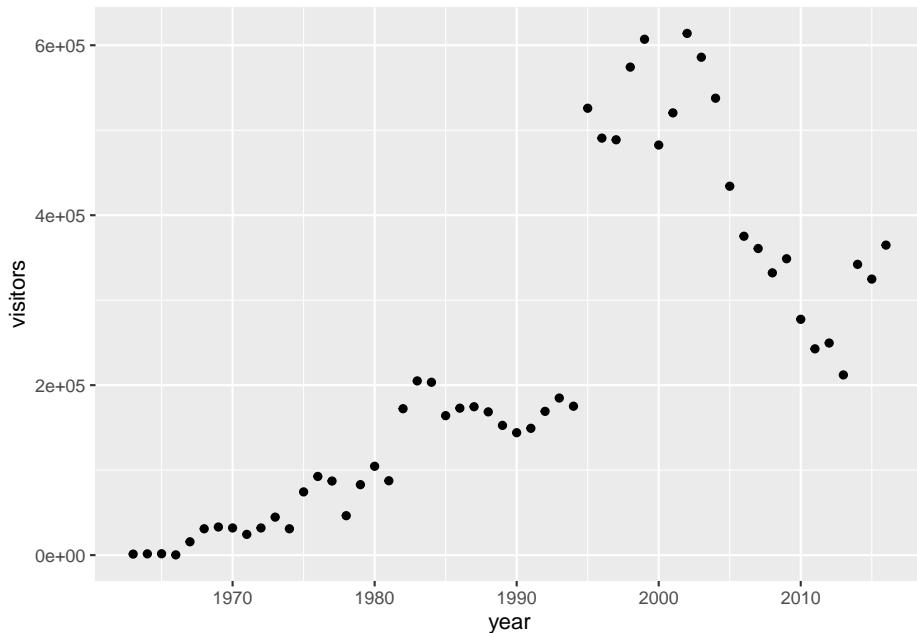
Let's do that to create a line graph of visitors to Channel Islands National Park:

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



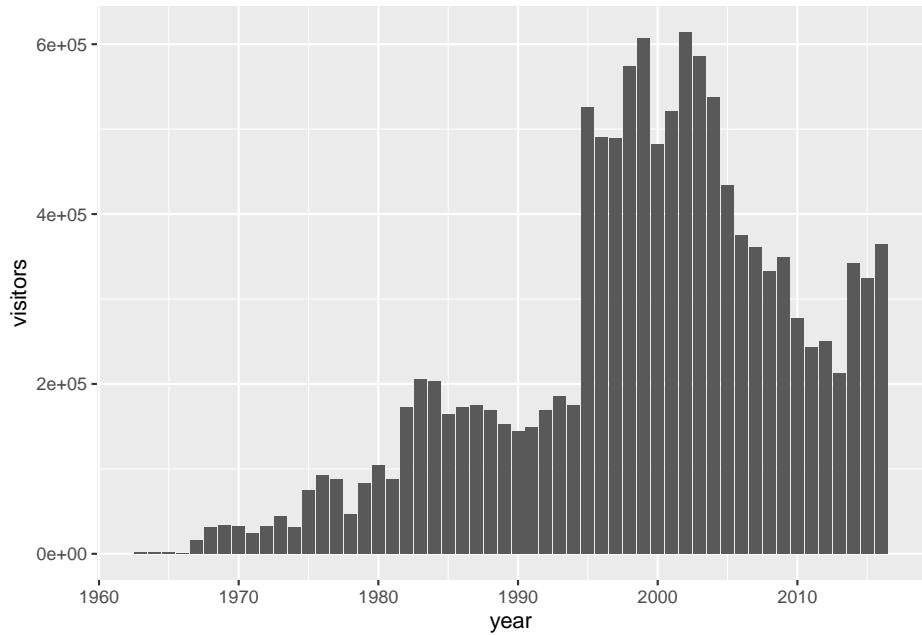
Or, we could change that to a scatterplot just by updating the `geom_*`:

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



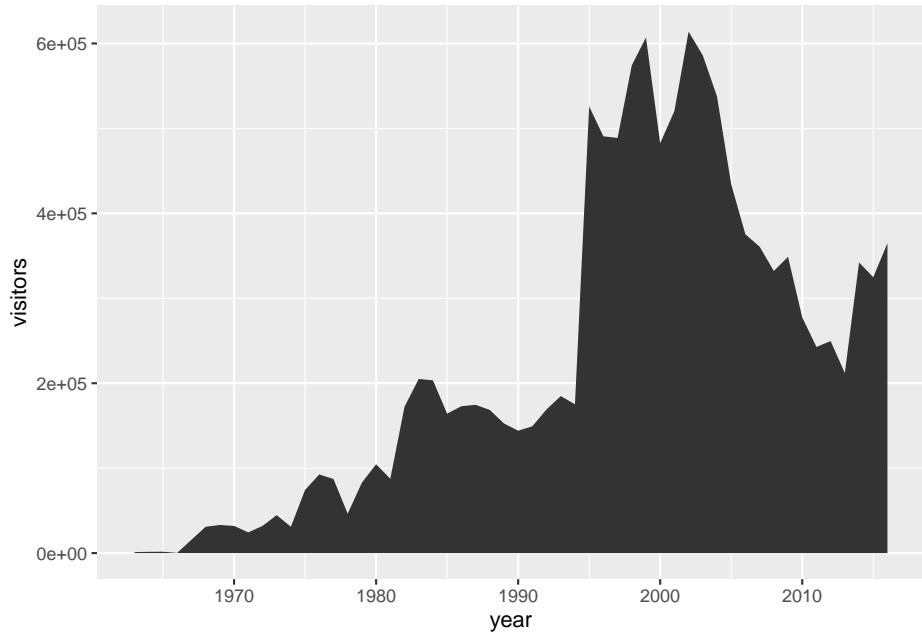
We could even do that for a column graph:

```
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 do some data viz customization.

5.2.3 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.

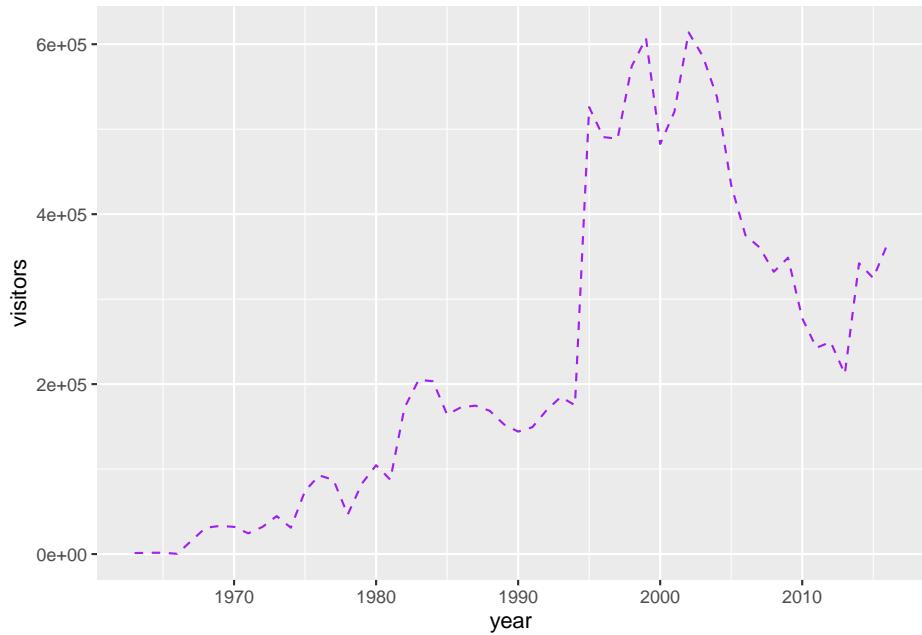
We can change the aesthetics of elements in a ggplot graph by adding arguments within the layer where that element is created.

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":

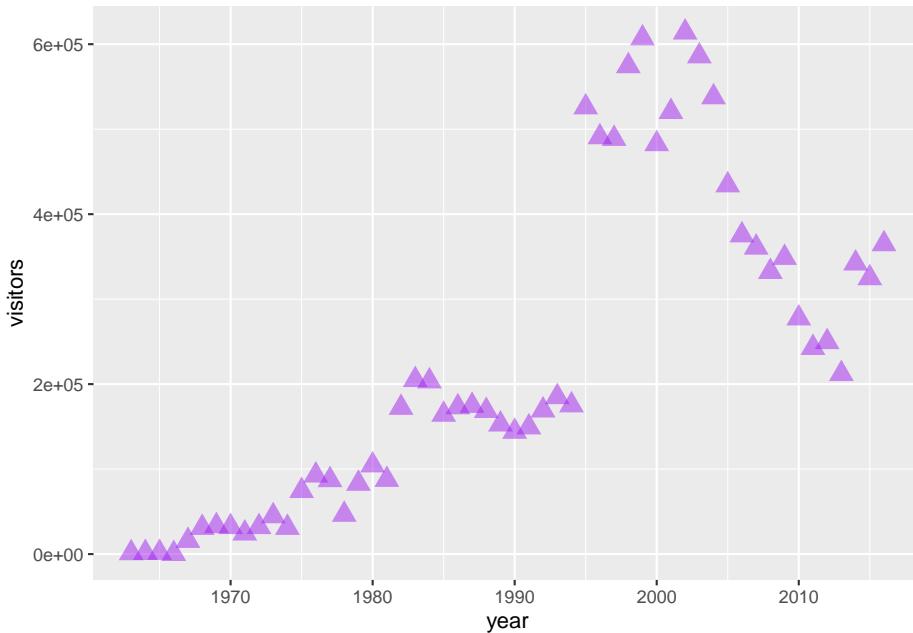
```
ggplot(data = ci_np, aes(x = year, y = visitors)) +
  geom_line(
    color = "purple",
    linetype = "dashed"
  )
```



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)
```



5.2.3.1 Activity: customize your own ggplot graph

Update one of the example graphs you created above to customize **at least** an element color and size!

5.2.4 Mapping variables onto aesthetics

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. To do that, we'll **map variables onto graph aesthetics**, meaning we'll change how an element on the graph looks based on a variable characteristic (usually, character or value).

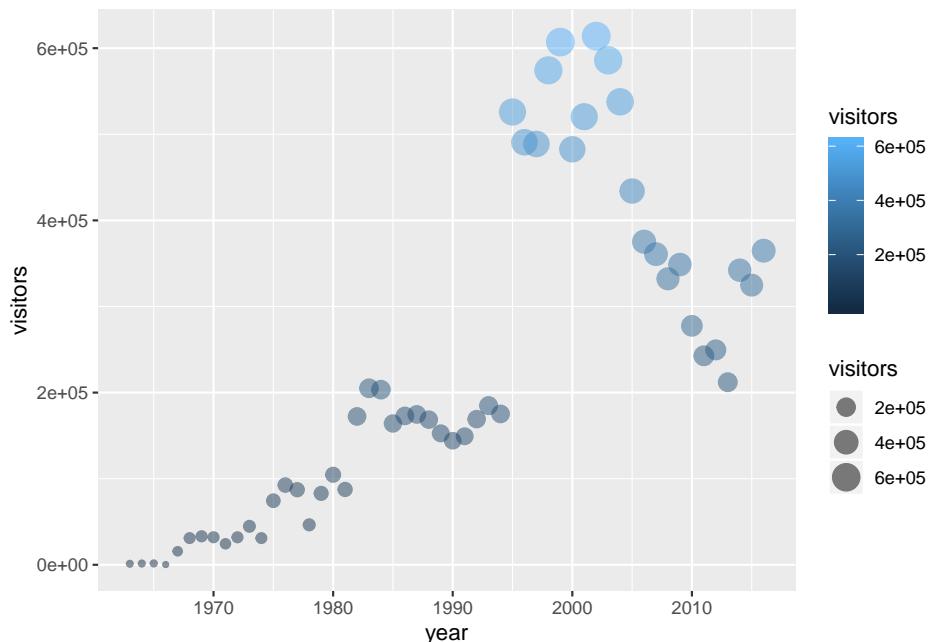
When we want to customize a graph element based on a variable's characteristic or value, add the argument within `aes()` in the appropriate `geom_*`() layer

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
  )
```

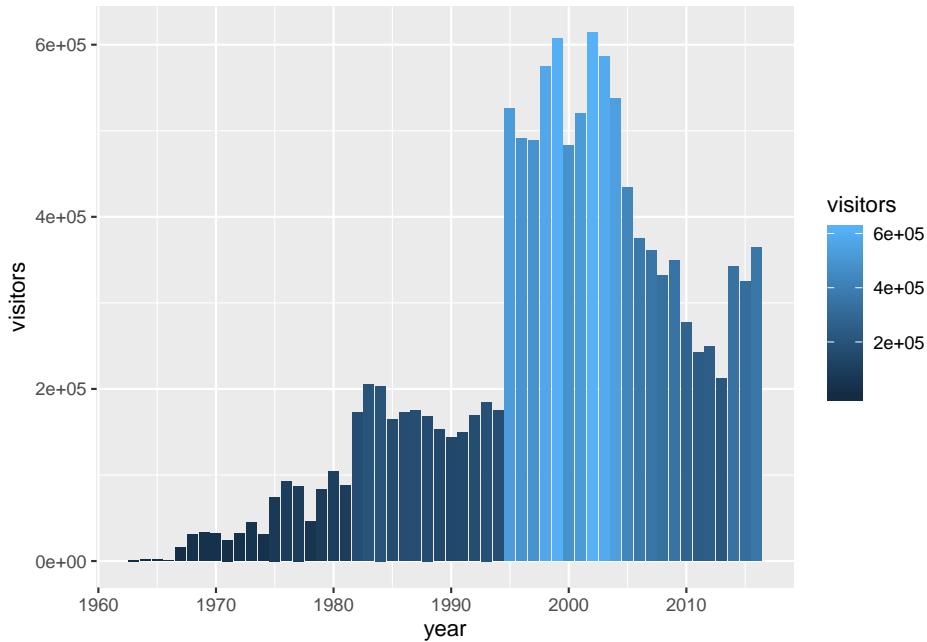


In the example above, notice that the two arguments that **do** depend on variables are within `aes()`, but since `alpha = 0.5` doesn't depend on a variable then it is *outside the `aes()` but still within the `geom_point()` layer*.

5.2.4.1 Activity: map variables onto graph aesthetics

Create a column plot of Channel Islands National Park visitation over time, where the **fill color** (argument: `fill =`) changes based on the number of **visitors**.

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



Sync your project with your GitHub repo.

5.2.5 ggplot2 complete themes

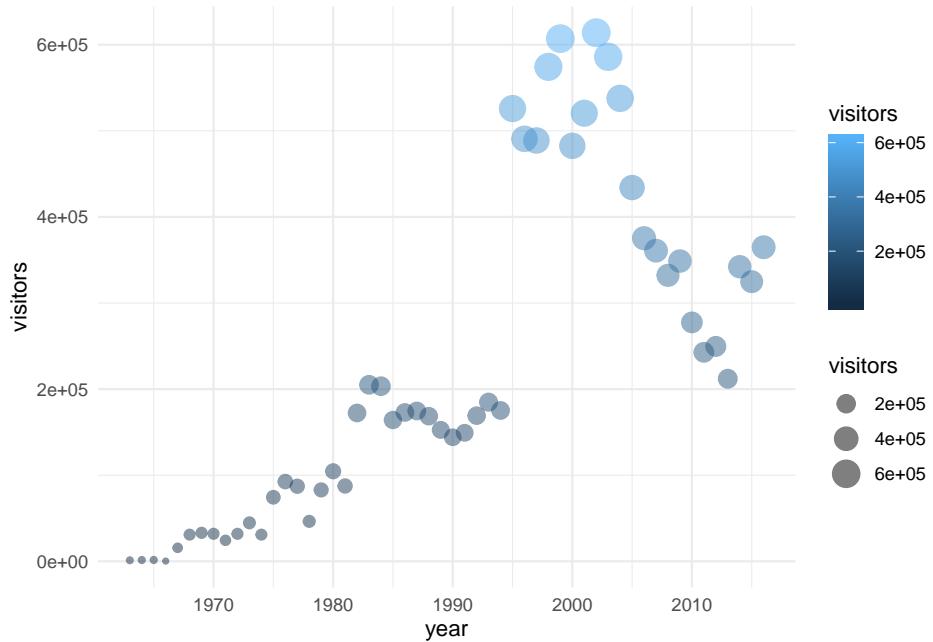
While every element of a ggplot graph is manually customizable, there are also built-in themes (`theme_*`()) that you can add to your ggplot code to make some major headway before making smaller tweaks manually.

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()`

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

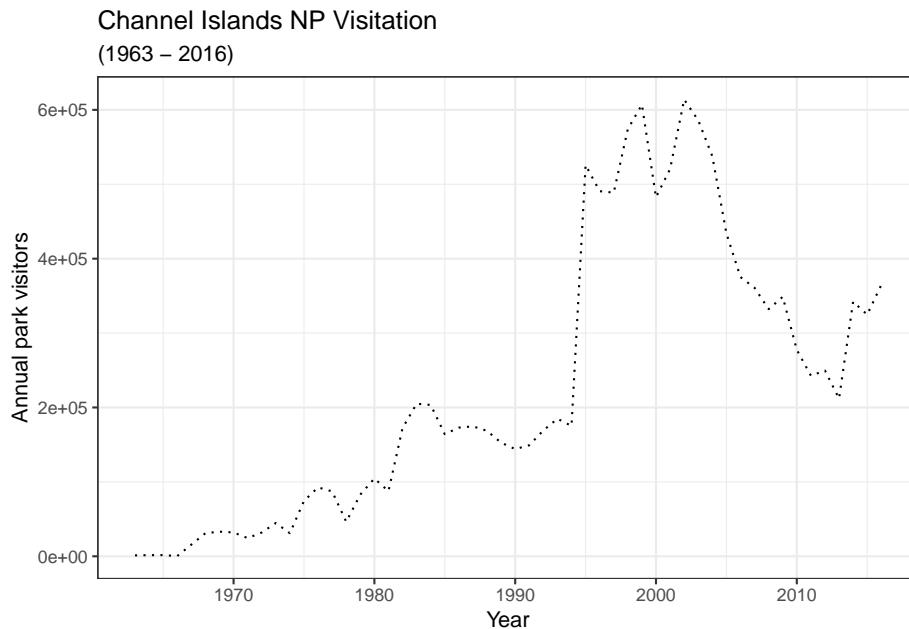
```
ggplot(data = ci_np, aes(x = year, y = visitors)) +
  geom_point(
    aes(size = visitors,
        color = visitors),
    alpha = 0.5
  ) +
  theme_minimal()
```



5.2.6 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)"
  )
```



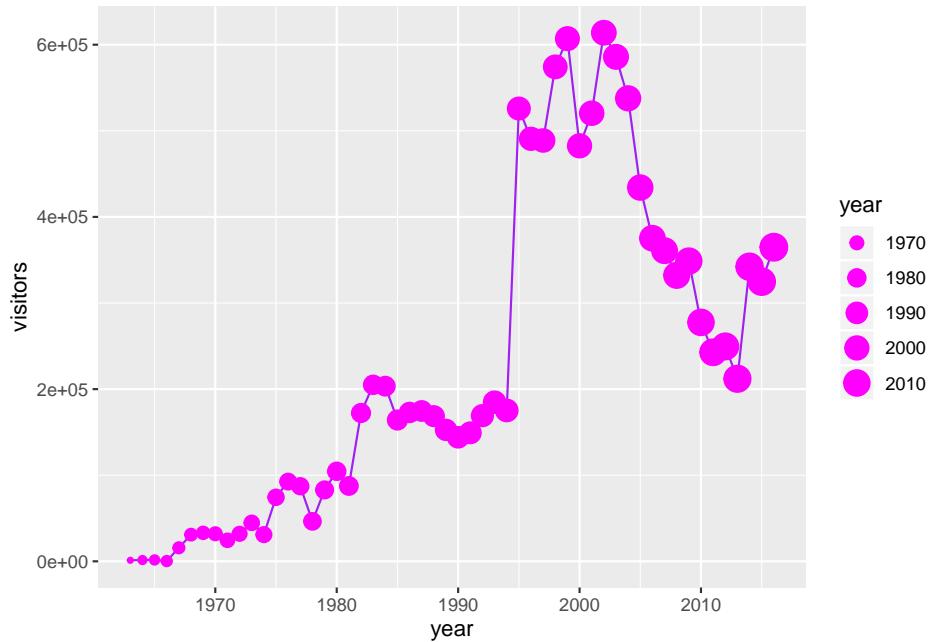
Note: If you want to update the formatting of axis values (for example, to convert to comma format instead of scientific format above), you can use the `scales` package options (see more from the R Cookbook).

5.2.7 Combining compatible geoms

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

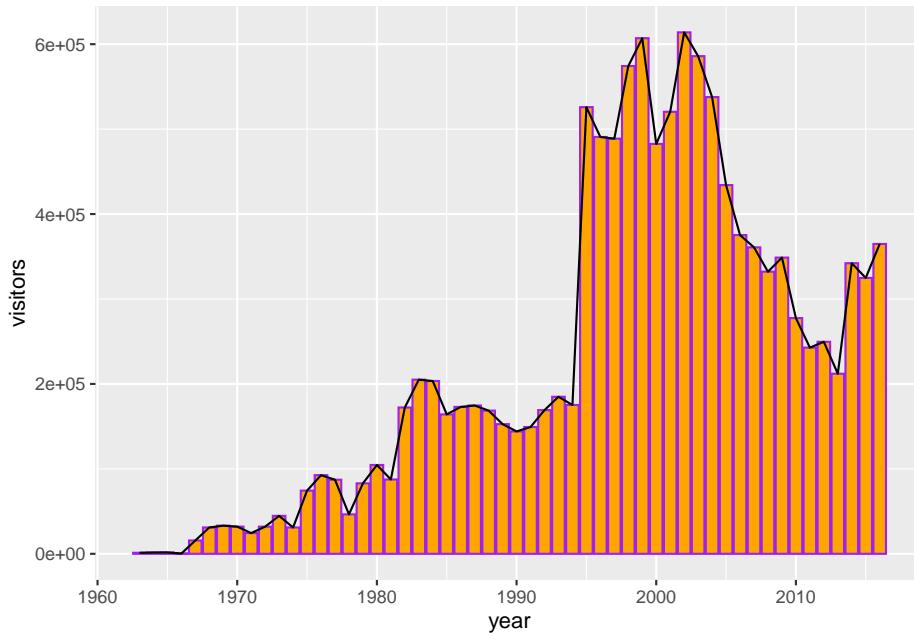
For example, adding points to a line graph:

```
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()
```

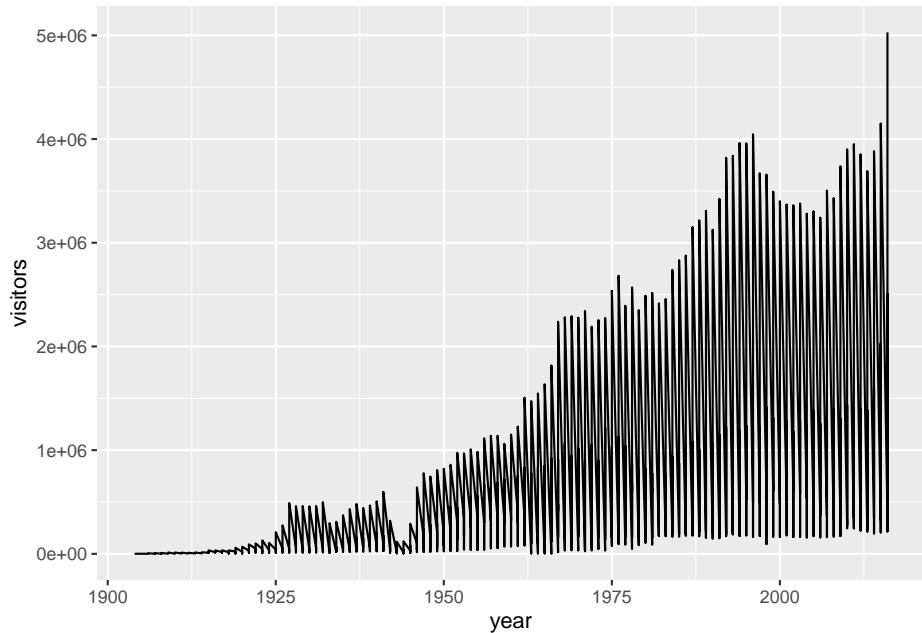


5.2.8 Multi-series ggplot graphs

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?

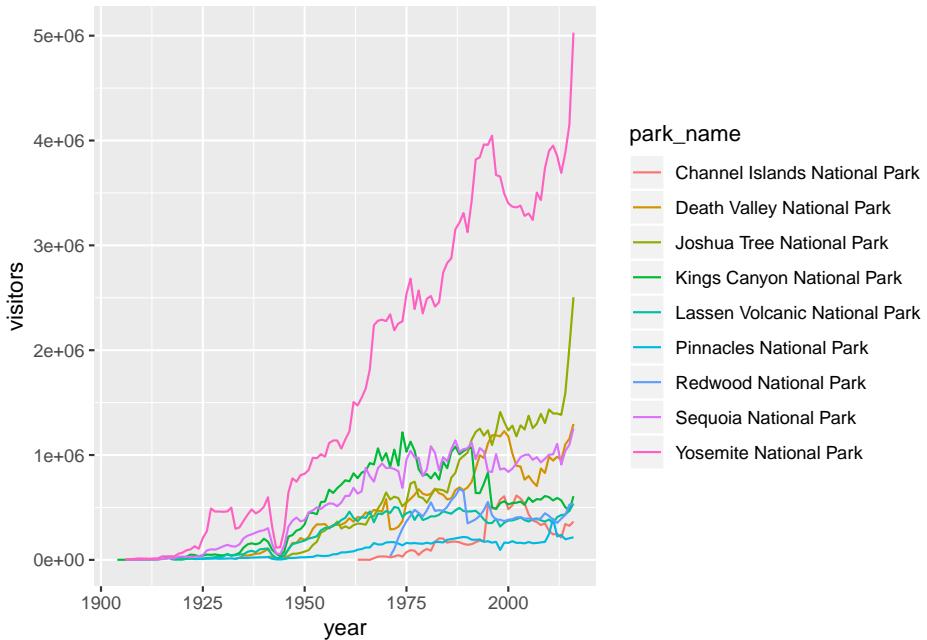
```
ggplot(data = ca_np, aes(x = year, y = visitors)) +  
  geom_line()
```



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()
```



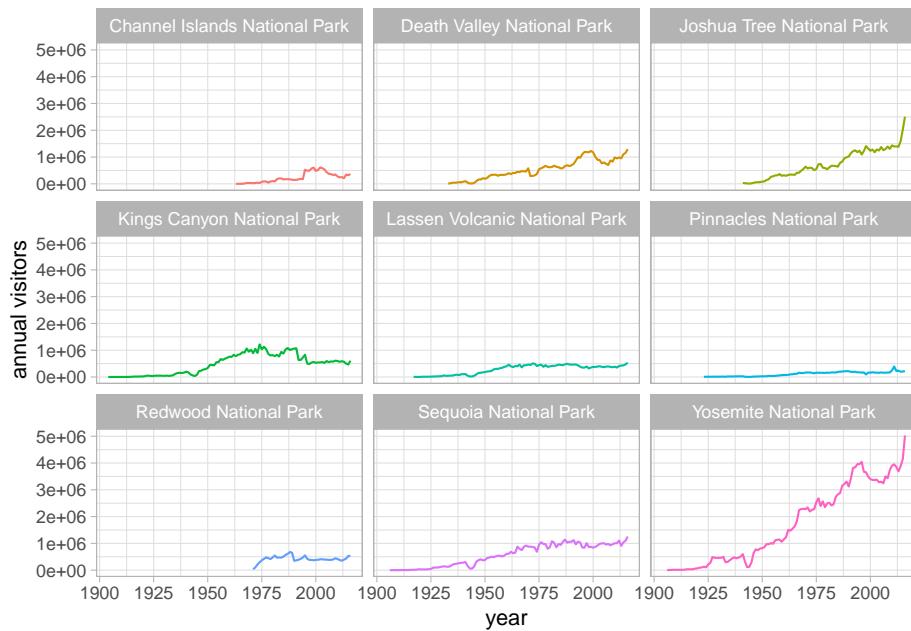
Note: You could also add that aesthetic (`color = park_name`) in the `geom_line()` layer, instead of in the topmost `ggplot()` layer.

5.2.9 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)
```



5.2.10 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:

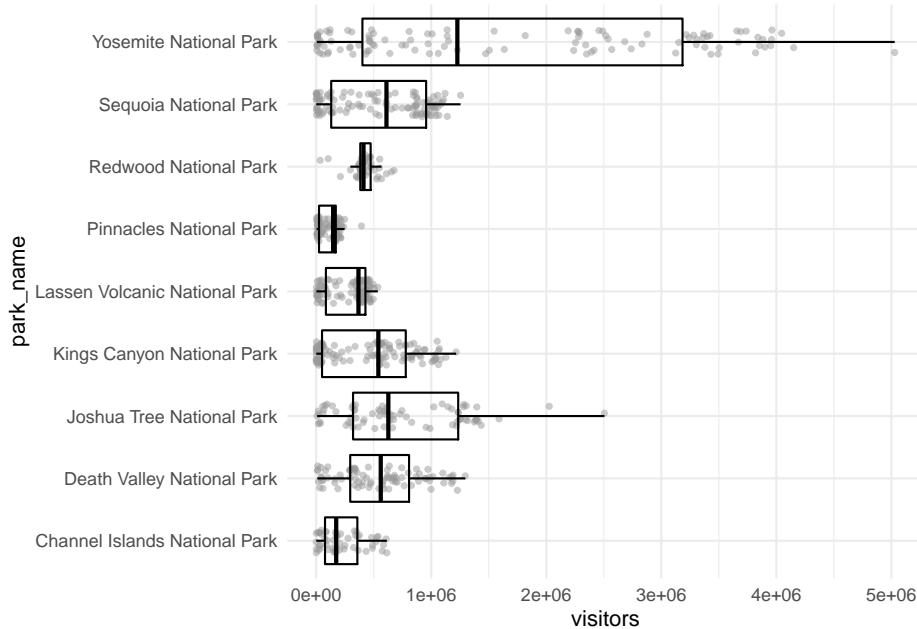
```
ggsave(here("figures", "np_graph.jpg"), dpi = 180, width = 8, height = 7)
```

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).

5.2.11 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()
```



Sync your project with your GitHub repo.

Chapter 6

dplyr and Pivot Tables

6.1 Summary

Pivot tables are powerful tools in Excel for summarizing data in different ways. We will create these tables using the `group_by` and `summarize` functions from the `dplyr` package (part of the Tidyverse). We will also learn how to format tables and practice creating a reproducible report using RMarkdown and sharing it with GitHub.

6.2 Objectives

In R, we can use `dplyr` for pivot tables by using 2 main verbs in combination: `group_by` and `summarize`. We will also continue to emphasize reproducibility in all our analyses.

- Discuss pivot tables in Excel
- Introduce `group_by()` `%>%` `summarize()` from the `dplyr` package
- Practice our reproducible workflow with RMarkdown and GitHub

6.3 Resources

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

6.4 Pivot table overview

Wikipedia describes a pivot table as a “table of statistics that summarizes the data of a more extensive table...This summary might include sums, averages, or other statistics, which the pivot table groups together in a meaningful way.” Fun fact: it also says that “Although pivot table is a generic term, Microsoft trademarked PivotTable in the United States in 1994.”

Pivot tables are a really powerful tool for summarizing data, and we can have similar functionality in R — as well as nicely automating and reporting these tables. We will learn about this using data about lobsters and will go back and forth between R and Excel as we learn.

Let’s start off in R, and have a look at the data.

6.5 RMarkdown setup

Let’s start a new RMarkdown file in our repo, at the top-level (where it will be created by default in our Project). I’ll call mine `pivot_lobsters.Rmd`.

In the setup chunk, let’s attach our libraries and read in our lobster data. In addition to the `tidyverse` package we will also use the `skimr` package. You will have to install it, but don’t want it to be installed every time you write your code. The following is a nice convention for having the install instructions available (on the same line) as the `library()` call.

```
## attach libraries
library(tidyverse)
library(readxl)
library(here)
library(skimr) # install.packages('skimr')

## read in data
lobsters <- read_xlsx(here("data/lobsters.xlsx"))
```

Let’s add a code chunk and explore the data in a few ways.

```
# explore data
head(lobsters) # year and month as well as a column for date

## # A tibble: 6 x 7
##   year month date    site transect replicate size_mm
##   <dbl> <dbl> <chr>  <chr>   <dbl> <chr>      <dbl>
## 1  2012     8 8/20/12 ivee       3 A           70
## 2  2012     8 8/20/12 ivee       3 B           60
## 3  2012     8 8/20/12 ivee       3 B           65
## 4  2012     8 8/20/12 ivee       3 B           70
```

```
## 5 2012     8 8/20/12 ivee      3 B      85
## 6 2012     8 8/20/12 ivee      3 C      60
```

`head()` gives us a look at the first rows of the data (6 by default). I like this because I can see the column names and get a sense of the shape of the data. I can also see the class of each column (double or character)

In this data set, every row is a unique observation. This is called “uncounted” data; you’ll see there is no row for how many lobsters were seen because each row is an observation, or an “n of 1”.

```
# explore data
summary(lobsters)

##      year        month         date       site
## Min.   :2012   Min.   :8.000  Length:6366   Length:6366
## 1st Qu.:2015  1st Qu.:8.000  Class  :character  Class  :character
## Median  :2017  Median  :8.000  Mode   :character  Mode   :character
## Mean    :2016  Mean    :8.017
## 3rd Qu.:2018  3rd Qu.:8.000
## Max.    :2018  Max.    :9.000
##
##      transect      replicate      size_mm
## Min.   :1.000  Length:6366      Min.   : 18.00
## 1st Qu.:2.000  Class  :character  1st Qu.: 65.00
## Median  :4.000  Mode   :character  Median  : 75.00
## Mean    :3.806
## 3rd Qu.:5.000
## Max.    :9.000
## NA's    :6
```

`summary` gives us summary statistics for each variable (column). I like this for numeric columns, but it doesn’t give a lot of useful information for non-numeric data. To have a look there I like using the `skimr` package:

```
# explore data
skimr::skim(lobsters)
```

Data summary

Name

lobsters

Number of rows

6366

Number of columns

Column type frequency:

character

3

numeric

4

Group variables

None

Variable type: character

skim_variable

n_missing

complete_rate

min

max

empty

n_unique

whitespace

date

0

1

6

7

0

38

0

site

0

1

4

4

```
0  
5  
0  
replicate  
0  
1  
1  
1  
0  
4  
0  
Variable type: numeric  
skim_variable  
n_missing  
complete_rate  
mean  
sd  
p0  
p25  
p50  
p75  
p100  
hist  
year  
0  
1  
2016.24  
1.66  
2012  
2015  
2017
```

2018

2018

month

0

1

8.02

0.13

8

8

8

8

9

transect

0

1

3.81

2.17

1

2

4

5

9

size_mm

6

1

73.02

13.61

18

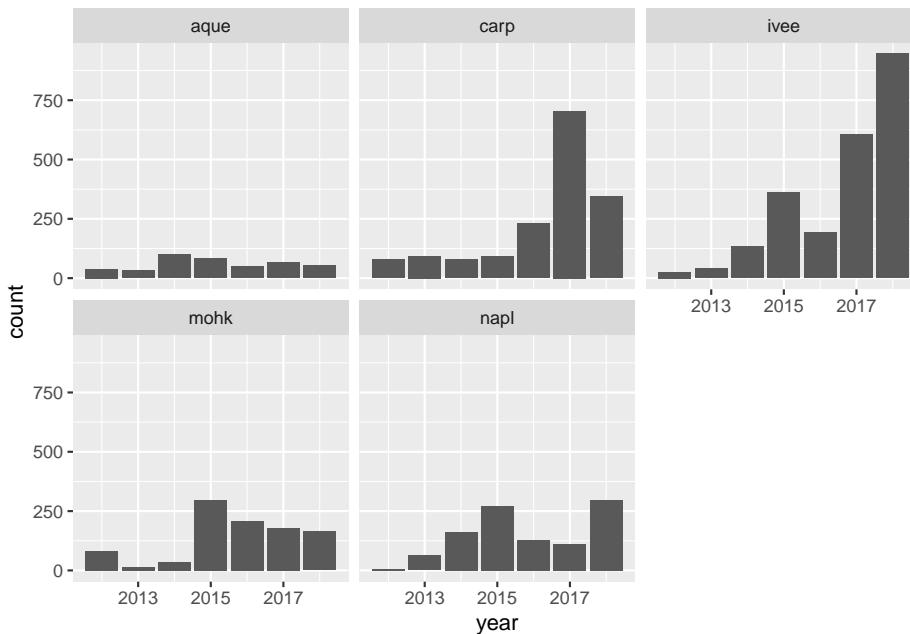
```
65
75
82
183
```

This `skimr::` notation is a reminder to me that `skim` is from the `skimr` package. It is a nice convention: it's a reminder to others (especially you!).

`skim` lets us look more at each variable. I particularly like looking at missing data. There are 6 missing values in the `size_mm` variable.

We can also make a quick plot to have a look at these data, and use our new `ggplot2` skills. Let's make a bar chart by year for each site

```
ggplot(lobsters, aes(x = year)) +
  geom_bar() +
  facet_wrap(~site)
```



(`geom_bar()` counts things and `geom_col()` is for values within the data (mean))

6.5.1 Our task

So this is all great to get a quick look. But what if we needed to report to someone about how the average size of lobsters has changed over time across

sites?

To answer this we need to do a pivot table in Excel, or data wrangling in R.

Let's start by having a quick look at what pivot tables can do in Excel.

6.6 Pivot table demo

Let's make a pivot table with our lobster data.

Let's start off with how many lobsters were counted each year. I want a count of rows by year.

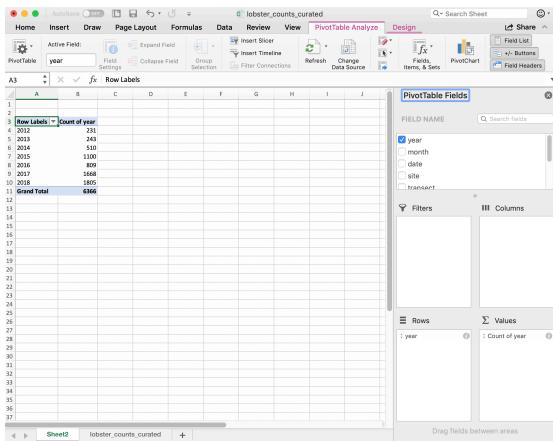
So to do this in Excel we would initiate the Pivot Table Process:

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

And then we'll get a little wizard to help us create the Pivot Table.

6.6.1 pivot one variable

I want to summarize by year, so I drag “year” down into the “Rows” box, and to get the counts by year I actually drag the same variable, “year” into the “Values” box. And it will create a Pivot Table for me! But “sum” as the default summary statistic, so I can click the little “I” icon to change this to count.



A few things to note:

- The pivot table is separate entity from our data (it's on a different sheet); the original data has not been affected
- The pivot table only shows the variables we requested; we don't see other columns (like date, month, or site).
- Notice that in Excel we retain the overall totals for each site (in bold, on the same line with the site name). This is nice for communicating about data. But it can be problematic for further analyses, because it could be easy to take a total of this column and introduce errors.

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

6.6.2 pivot two variables

We can also add site as a second variable by dragging it:

	A	B	C	D	E	F	G
3	Row Labels	Count of year					
4	2012	422					
5	aqua	38					
6	carp	32					
7	iver	78					
8	mohk	83					
9	nopl	6					
10	2013	243					
11	aqua	32					
12	carp	93					
13	iver	40					
14	mohk	15					
15	nopl	53					
16	2014	510					
17	aqua	100					
18	carp	79					
19	iver	132					
20	mohk	90					
21	nopl	143					
22	2015	1100					
23	aqua	63					
24	carp	90					
25	iver	361					
26	mohk	295					
27	nopl	270					
28	2016	869					
29	aqua	42					
30	carp	231					
31	iver	193					
32	mohk	193					
33	nopl	127					
34	2017	1648					
35	aqua	67					
36	carp	705					
37	iver	200					
38	mohk	178					
39	nopl	112					
40	2018	1603					
41	aqua	54					
42	carp	343					
43	iver	166					
44	mohk	164					
45	nopl	298					
46	Grand Total	6366					
47							

And then can reverse the order by dragging:

	A	B	C	D	E	F	G
3	Row Labels	Count of year					
4	2012	422					
5	aqua	38					
6	2013	32					
7	2014	100					
8	2015	83					
9	2016	48					
10	2017	67					
11	2018	54					
12	carp	1619					
13	2012	78					
14	2013	93					
15	2014	79					
16	2015	20					
17	2016	231					
18	2017	705					
19	2018	343					
20	iver	2304					
21	2012	26					
22	2013	40					
23	2014	132					
24	2015	361					
25	2016	193					
26	2017	608					
27	2018	966					
28	mohk	982					
29	2012	83					
30	2013	15					
31	2014	36					
32	2015	296					
33	2016	210					
34	2017	178					
35	2018	164					
36	nopl	1029					
37	2012	6					
38	2013	63					
39	2014	183					
40	2015	270					
41	2016	127					
42	2017	112					
43	2018	298					
44	Grand Total	6366					
45							
46							
47							

So in terms of our final interest of average size by site and year, we are on our way! I'm going to stop here because we want to be able to do this in R.

The power of R is in the automation, and in keeping that raw data truly raw.

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

6.7 group_by() %>% summarize()

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

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

I think it’s incredibly powerful to visualize what we are talking about with our data when do do these kinds of operations. It looks like this (from RStudio’s cheatsheet; all cheatsheets available from <https://rstudio.com/resources/cheatsheets>):



When we were reporting by year or site, we were essentially modifying what we were grouping by (the different colors here in this figure).

Let’s do this in R.

6.7.1 group_by one variable

Let’s try this on our `lobsters` data, just like we did in Excel. We will count the the total number of lobster by year. In R vocabulary, we will `group_by` year and then `summarize` by counting using `n()`, which is a function from `dplyr`. `n()` counts the number of times an observation shows up, and since this is uncounted data, this will count each row. We’ll also use the pipe operator `%>%`, which you can read as “and then”.

This to me reads: “take the lobsters data and then group_by year and then summarize by count in a new column called ‘count’”

```
lobsters %>%
  group_by(year) %>%
  summarize(count_by_year = n())
```

```
## # A tibble: 7 x 2
##   year  count_by_year
##   <dbl>      <int>
## 1 2012       231
## 2 2013       243
```

```
## 3 2014      510
## 4 2015     1100
## 5 2016      809
## 6 2017    1668
## 7 2018    1805
```

Notice how together, `group_by` and `summarize` minimize the amount of information we see. We also saw this with the pivot table. We lose the other columns that aren't involved here.

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

```
lobsters %>%
  summarize(count = n())
```

```
## # A tibble: 1 x 1
##   count
##   <int>
## 1 6366
```

So if we don't `group_by` first, we will get a single summary statistic (sum in this case) for the whole dataset.

Another question: what if we *only* group_by?

```
lobsters %>%
  group_by(year)
```

```
## # A tibble: 6,366 x 7
## # Groups:   year [7]
##   year month date   site transect replicate size_mm
##   <dbl> <dbl> <chr>  <chr>   <dbl> <chr>      <dbl>
## 1 2012     8 8/20/12 ivee      3 A        70
## 2 2012     8 8/20/12 ivee      3 B        60
## 3 2012     8 8/20/12 ivee      3 B        65
## 4 2012     8 8/20/12 ivee      3 B        70
## 5 2012     8 8/20/12 ivee      3 B        85
## 6 2012     8 8/20/12 ivee      3 C        60
## 7 2012     8 8/20/12 ivee      3 C        65
## 8 2012     8 8/20/12 ivee      3 C        67
## 9 2012     8 8/20/12 ivee      3 D        70
## 10 2012    8 8/20/12 ivee      4 B        85
## # ... with 6,356 more rows
```

6.7.2 RStudio Viewer

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

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

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

6.7.3 group_by multiple variables

Great. Now let's summarize by both year and site like we did in the pivot table. We are able to `group_by` more than one variable. Let's do this together:

```
lobsters %>%
  group_by(site, year) %>%
  summarize(count_by_siteyear = n())

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

text.

6.7.4 summarize multiple variables

We can summarize multiple variables at a time. So far we've done the count of lobster observations. Let's also do the mean and standard deviation. First let's use the `mean()` function to calculate the mean. We do this within the same `summarize()` function, but we can add a new line to make it easier to read. Notice how when you put your cursor within the parenthesis and hit return, the indentation will automatically align.

```
lobsters %>%
  group_by(site, year) %>%
  summarize(count_by_siteyear = n(),
            mean_size_mm = mean(size_mm))

## # A tibble: 35 x 4
## # Groups:   site [5]
##   site    year count_by_siteyear mean_size_mm
##   <chr> <dbl>             <int>        <dbl>
## 1 aque   2012              38          71
## 2 aque   2013              32          72.1
## 3 aque   2014             100          76.9
## 4 aque   2015              83          68.5
## 5 aque   2016              48          68.7
## 6 aque   2017              67          73.9
## 7 aque   2018              54          71.7
## 8 carp    2012              78          74.4
## 9 carp    2013              93          76.6
## 10 carp   2014             79          NA
## # ... with 25 more rows
```

Aside Command-I will properly indent selected lines.

Great! But this will actually calculate some of the means as NA because one or more values in that year are NA. So we can pass an argument that says to remove NAs first before calculating the average. Let's do that, and then also calculate the standard deviation with the `sd()` function:

```
lobsters %>%
  group_by(site, year) %>%
  summarize(count_by_siteyear = n(),
            mean_size_mm = mean(size_mm, na.rm=TRUE),
            sd_size_mm = sd(size_mm, na.rm=TRUE))

## # A tibble: 35 x 5
## # Groups:   site [5]
##   site    year count_by_siteyear mean_size_mm sd_size_mm
##   <chr> <dbl>             <int>        <dbl>       <dbl>
## 1 aque   2012              38          71         10.2
```

```

## 2 aque 2013      32    72.1   12.3
## 3 aque 2014     100    76.9   9.32
## 4 aque 2015      83    68.5   12.6
## 5 aque 2016      48    68.7   12.5
## 6 aque 2017      67    73.9   11.9
## 7 aque 2018      54    71.7   8.14
## 8 carp 2012      78    74.4   14.6
## 9 carp 2013      93    76.6   8.71
## 10 carp 2014     79    79.1   8.57
## # ... with 25 more rows

```

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

```

siteyear_summary <- lobsters %>%
  group_by(site, year) %>%
  summarize(count_by_siteyear = n(),
            mean_size_mm = mean(size_mm, na.rm = TRUE),
            sd_size_mm = sd(size_mm, na.rm = TRUE))

siteyear_summary

## # A tibble: 35 x 5
## # Groups:   site [5]
##   site   year count_by_siteyear mean_size_mm sd_size_mm
##   <chr> <dbl>           <int>        <dbl>       <dbl>
## 1 aque  2012            38         71          10.2
## 2 aque  2013            32         72.1        12.3
## 3 aque  2014           100        76.9        9.32
## 4 aque  2015            83         68.5        12.6
## 5 aque  2016            48         68.7        12.5
## 6 aque  2017            67         73.9        11.9
## 7 aque  2018            54         71.7        8.14
## 8 carp  2012            78         74.4        14.6
## 9 carp  2013            93         76.6        8.71
## 10 carp 2014            79         79.1        8.57
## # ... with 25 more rows

```

6.7.5 Activity

1. Calculate the median `size_mm` (Hint: `?median`) and
2. create and `ggsave()` a plot.

Then, save, commit, and push your .Rmd, .html, and .png.

Solution (no peeking):

```
siteyear_summary <- lobsters %>%
  group_by(site, year) %>%
  summarise(count_by_siteyear = n(),
            mean_size_mm = mean(size_mm, na.rm = TRUE),
            sd_size_mm = sd(size_mm, na.rm = TRUE),
            median_size_mm = median(size_mm, na.rm = TRUE))

siteyear_summary

## a ggplot option:
ggplot(data = siteyear_summary, aes(x = year, y = median_size_mm, color = site)) +
  geom_line()
ggsave(here("figures", "lobsters-line.png"))

## another option:
ggplot(siteyear_summary, aes(x = year, y = median_size_mm)) +
  geom_col() +
  facet_wrap(~site)
ggsave(here("figures", "lobsters-col.png"))
```

Don't forget to knit, commit, and push!

Nice work everybody.

6.8 Oh no, our colleague sent the wrong data!

Oh no! After all our analyses and everything we've done, our colleague just emailed us at 4:30pm on Friday that he sent the wrong data and we need to redo all our analyses with a new .xlsx file: `lobsters2.xlsx`, not `lobsters.xlsx`. Aaaaah!

If we were doing this in Excel, this would be a bummer; we'd have to rebuild our pivot table and click through all of our logic again. And then export our figures and save them into our report.

But, since we did it in R, we are much safer. We can go back to the top of our RMarkdown file, and read in the updated dataset, and then re-knit. We will still need to check that everything outputs correctly, (and that column headers haven't been renamed), but our first pass will be to update the filename and re-knit:

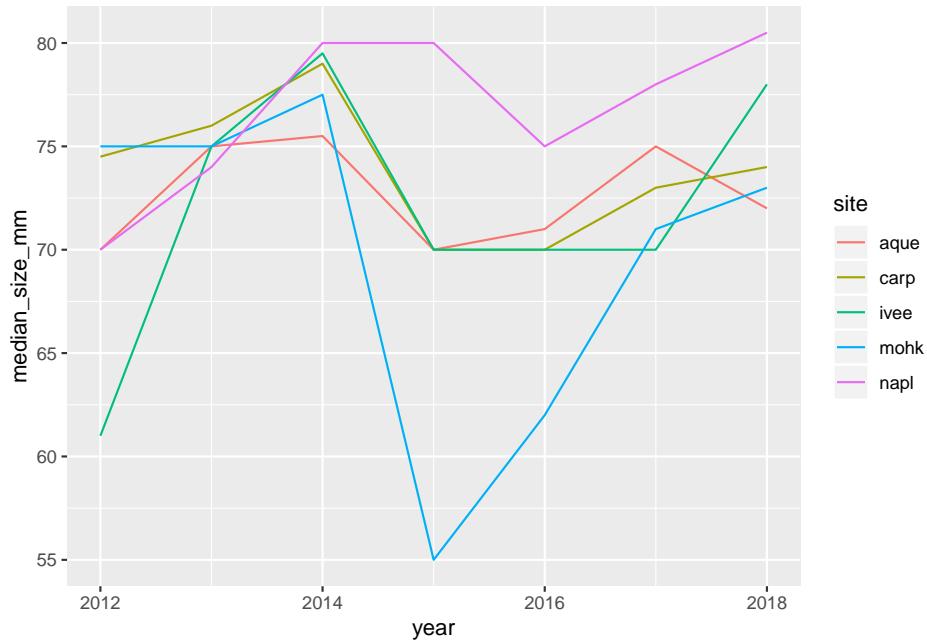
```
## read in data
lobsters <- read_xlsx(here("data/lobsters2.xlsx"))
```

And now we can see that our plot updated as well:

```
siteyear_summary <- lobsters %>%
  group_by(site, year) %>%
  summarise(count_by_siteyear = n(),
            mean_size_mm = mean(size_mm, na.rm = TRUE),
            sd_size_mm = sd(size_mm, na.rm = TRUE),
            median_size_mm = median(size_mm, na.rm = TRUE), )

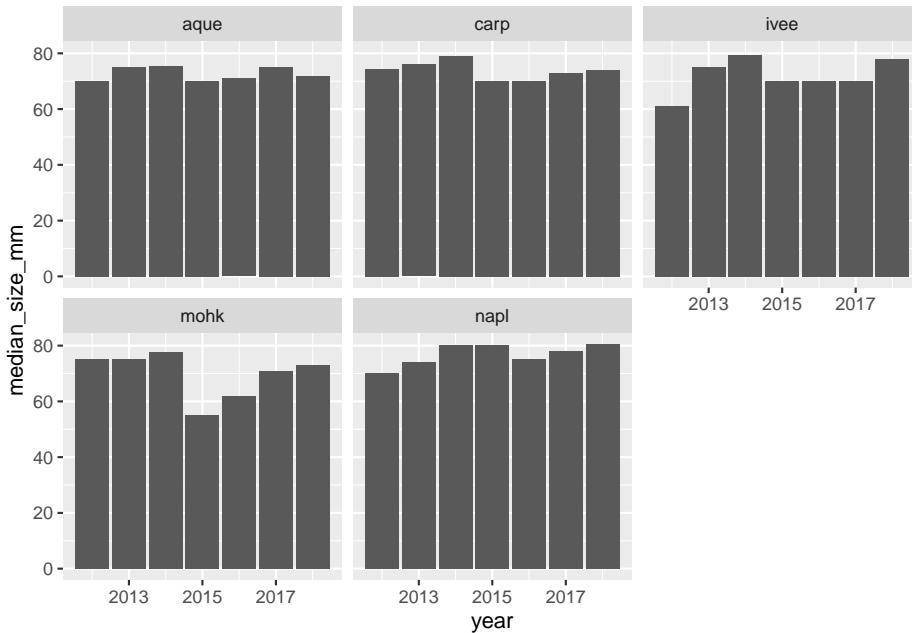
siteyear_summary

## # A tibble: 35 x 6
## # Groups:   site [5]
##   site   year count_by_siteyear mean_size_mm sd_size_mm median_size_mm
##   <chr> <dbl>           <int>        <dbl>       <dbl>        <dbl>
## 1 aque   2012            38          71         10.2        70
## 2 aque   2013            32          72.1        12.3        75
## 3 aque   2014           100          76.9        9.32       75.5
## 4 aque   2015            83          68.5        12.6        70
## 5 aque   2016            48          68.7        12.5        71
## 6 aque   2017            67          73.9        11.9        75
## 7 aque   2018            54          71.7        8.14        72
## 8 carp    2012            78          74.4        14.6       74.5
## 9 carp    2013            93          76.6        8.71        76
## 10 carp   2014            79          79.1        8.57        79
## # ... with 25 more rows
## # a ggplot option:
ggplot(data = siteyear_summary, aes(x = year, y = median_size_mm, color = site)) +
  geom_line()
```



```
ggsave(here("figures", "lobsters-line.png"))
```

```
## Saving 6.5 x 4.5 in image
## another option:
ggplot(siteyear_summary, aes(x = year, y = median_size_mm)) +
  geom_col() +
  facet_wrap(~site)
```



```
ggsave(here("figures", "lobsters-col.png"))
```

```
## Saving 6.5 x 4.5 in image
```

6.8.1 Knit, push, & show differences on GitHub

So cool.

6.8.2 dplyr::count()

Now that we've spent time with group_by %>% summarize, there is a shortcut if you only want to summarize by count. This is with a function called `count()`, and it will group_by your selected variable, count, and then also ungroup. It looks like this:

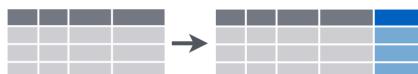
```
lobsters %>%
  count(site, year)

## This is the same as:
lobsters %>%
  group_by(site, year) %>%
  summarize(n = n()) %>%
  ungroup()
```

Switching gears...

6.9 `mutate()`

Make New Variables



There are a lot of times where you don't want to summarize your data, but you do want to operate beyond the original data. This is often done by adding a column. We do this with the `mutate()` function from `dplyr`. Let's try this with our original lobsters data. The sizes are in millimeters but let's say it was important for them to be in meters. We can add a column with this calculation:

```
# quick reminder what this looks like
head(lobsters)
```

```
## # A tibble: 6 x 7
##   year month date    site transect replicate size_mm
##   <dbl> <dbl> <chr>  <chr>   <dbl> <chr>     <dbl>
## 1 2012     8 8/20/12 ivee      3 A         70
## 2 2012     8 8/20/12 ivee      3 B         60
## 3 2012     8 8/20/12 ivee      3 B         65
## 4 2012     8 8/20/12 ivee      3 B         70
## 5 2012     8 8/20/12 ivee      3 B         85
## 6 2012     8 8/20/12 ivee      3 C         60
```

```
lobsters %>%
  mutate(size_m = size_mm / 1000)
```

```
## # A tibble: 6,366 x 8
##   year month date    site transect replicate size_mm size_m
##   <dbl> <dbl> <chr>  <chr>   <dbl> <chr>     <dbl>  <dbl>
## 1 2012     8 8/20/12 ivee      3 A         70  0.07
## 2 2012     8 8/20/12 ivee      3 B         60  0.06
## 3 2012     8 8/20/12 ivee      3 B         65  0.065
## 4 2012     8 8/20/12 ivee      3 B         70  0.07
## 5 2012     8 8/20/12 ivee      3 B         85  0.085
## 6 2012     8 8/20/12 ivee      3 C         60  0.06
## 7 2012     8 8/20/12 ivee      3 C         65  0.065
## 8 2012     8 8/20/12 ivee      3 C         67  0.067
## 9 2012     8 8/20/12 ivee      3 D         70  0.07
## 10 2012    8 8/20/12 ivee      4 B         85  0.085
```

```
## # ... with 6,356 more rows
```

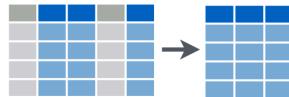
If we want to add a column that has the same value repeated, we can pass it just one value, either a number or a character string (in quotes). And let's save this as a variable called `lobsters_detailed`

```
lobsters_detailed <- lobsters %>%
  mutate(size_m = size_mm / 1000,
        millenia = 2000,
        observer = "Allison Horst")
```

6.10 `select()`

We will end with one final function, `select`. This is how to choose, retain, and move your data by columns:

Subset Variables (Columns)



Let's say that we want to present this data finally with only columns for date, site, and size in meters. We would do this:

```
lobsters_detailed %>%
  select(date, site, size_m)

## # A tibble: 6,366 x 3
##   date    site  size_m
##   <chr>   <chr>  <dbl>
## 1 8/20/12 ivee   0.07
## 2 8/20/12 ivee   0.06
## 3 8/20/12 ivee   0.065
## 4 8/20/12 ivee   0.07
## 5 8/20/12 ivee   0.085
## 6 8/20/12 ivee   0.06
## 7 8/20/12 ivee   0.065
## 8 8/20/12 ivee   0.067
## 9 8/20/12 ivee   0.07
## 10 8/20/12 ivee   0.085
## # ... with 6,356 more rows
```

One last time, let's knit, save, commit, and push to GitHub.

6.11 Deep thoughts

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

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

6.12 Efficiency Tips

arrow keys with shift, option, command

Chapter 7

Tidying

TODO: janitor: adorn and kable

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()`

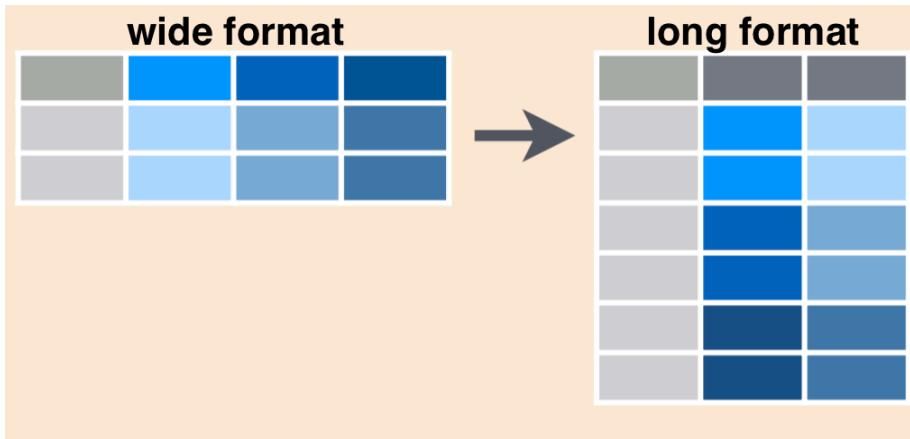
7.4.2 Reshaping with `tidyr::pivot_longer()` and `tidyr::pivot_wider()`

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 %>%
  tidyverse::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>      <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 `tidyr::unite()` and `tidyr::separate()`

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

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

```
id
genus
species
common_name
1
Scorpaena
guttata
```

```
sculpin
```

```
2
```

```
Sebastes
```

```
miniatus
```

```
vermillion
```

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

```
id
```

```
scientific_name
```

```
common_name
```

```
1
```

```
Scorpaena guttata
```

```
sculpin
```

```
2
```

```
Sebastes miniatus
```

```
vermillion
```

Or we may want to do the reverse (separate information from a single column into multiple columns). Here, we'll learn `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)

A diagram illustrating the function of `tidy::unite()`. It shows a transformation from a wide data frame to a long data frame. On the left, there is a grid of 12 squares arranged in 3 rows and 4 columns. The colors of the squares transition from grey to blue across the columns. An arrow points from this grid to a second grid on the right. The second grid has the same 3 rows but only 3 columns. The first two columns are composed of the first two squares from each row of the original grid, and the third column is composed of the last two squares from each row of the original grid. This visualizes how `tidy::unite()` combines multiple columns into a single column.

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 %>%
  tidy::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.

`tidy::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 %>%
  tidy::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.”

`tidy::separate(storms, date, c("y", "m", "d"))`

Separate one column into several.

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

```
inverts_sep <- inverts_triple %>%
  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 %>%
  tidyverse::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 `tidyverse::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>
## 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 (NAAs)

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:

- `count()`: get counts of observations for groupings we specify
- `mutate()`: **add** a new column, while keeping the existing ones
- `group_by()`: let R know that **groups** exist within the dataset, by variable(s)
- `summarize()`: calculate a value (that you specify) for each group, then report each group's value in a table

In this session, we'll expand our data wrangling toolkit using:

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

The combination of `filter()` and `*_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.

8.1.1 Objectives

- Use `filter()` to subset data frames, returning **rows** that satisfy variable conditions
- Use `full_join()`, `left_join()`, and `inner_join()` to merge data frames, with different endpoints in mind
- Use `filter()` and `*_join()` as part of a wrangling sequence

8.1.2 Resources

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

8.2 Lessons

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

Create a new R Markdown document in your r-workshop project and knit to save as **filter_join.Rmd**. Remove all the example code (everything below the set-up code chunk).

In this session, we'll use three packages:

- `tidyverse`
- `readxl`
- `here`

Attach the packages in the setup code chunk in your .Rmd:

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

Then create a new code chunk to read in three files from your ‘data’ subfolder:

- `invert_counts_curated.xlsx`
- `fish_counts_curated.csv`
- `kelp_counts.xlsx`

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

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

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

8.2.2 `filter()` to conditionally subset by rows

Use `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.

Subset Observations (Rows)



Some examples in words that might inspire you to use `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 `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

Here, we’ll learn some common ways to use `filter()`.

8.2.2.1 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_garibaldi <- filter(fish_counts, common_name == "garibaldi")
```

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

You could also do this using the pipe operator `%>%`:

```
fish_garibaldi <- fish_counts %>%
  filter(common_name == "garibaldi")
```

8.2.2.2 Activity

Task: Create a subset from the `fish_counts` data frame, stored as object `fish_abur`, that only contains observations from Arroyo Burro (site ‘abur’).

Solution:

```
fish_abur <- fish_counts %>%
  filter(site == "abur")
```

Explore the subset you just created to ensure that only Arroyo Burro observations are returned.

8.2.2.3 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 <- filter(fish_counts, total_count >= 50)
```

Or, using the pipe:

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

8.2.2.4 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 %>%
  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 %>%
  filter(common_name == "garibaldi" | common_name == "blacksmith" | common_name == "bl
```

8.2.2.5 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 `filter()`, add a comma (or ampersand, &) between arguments for multiple *AND* conditions:

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

Or you could just do two filter steps in sequence:

```
# Written as sequential filter steps:

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

8.2.2.5.1 Activity: combined filter conditions

Task: Create a subset from the `fish_counts` data frame, called `low_gb_wr` that only contains:

- Observations of *garibaldi* and *rock wrasse*
- Where the `total_count` is *less than or equal to 10*

Solution:

```
low_gb_wr <- fish_counts %>%
  filter(common_name %in% c("garibaldi", "rock wrasse"),
         total_count <= 10)
```

Sync your local project to your repo on GitHub.

8.2.2.6 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 %>%
  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 %>%
  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 %>%
  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
```

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

We can also use `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:

- Converts to long(er) 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
```

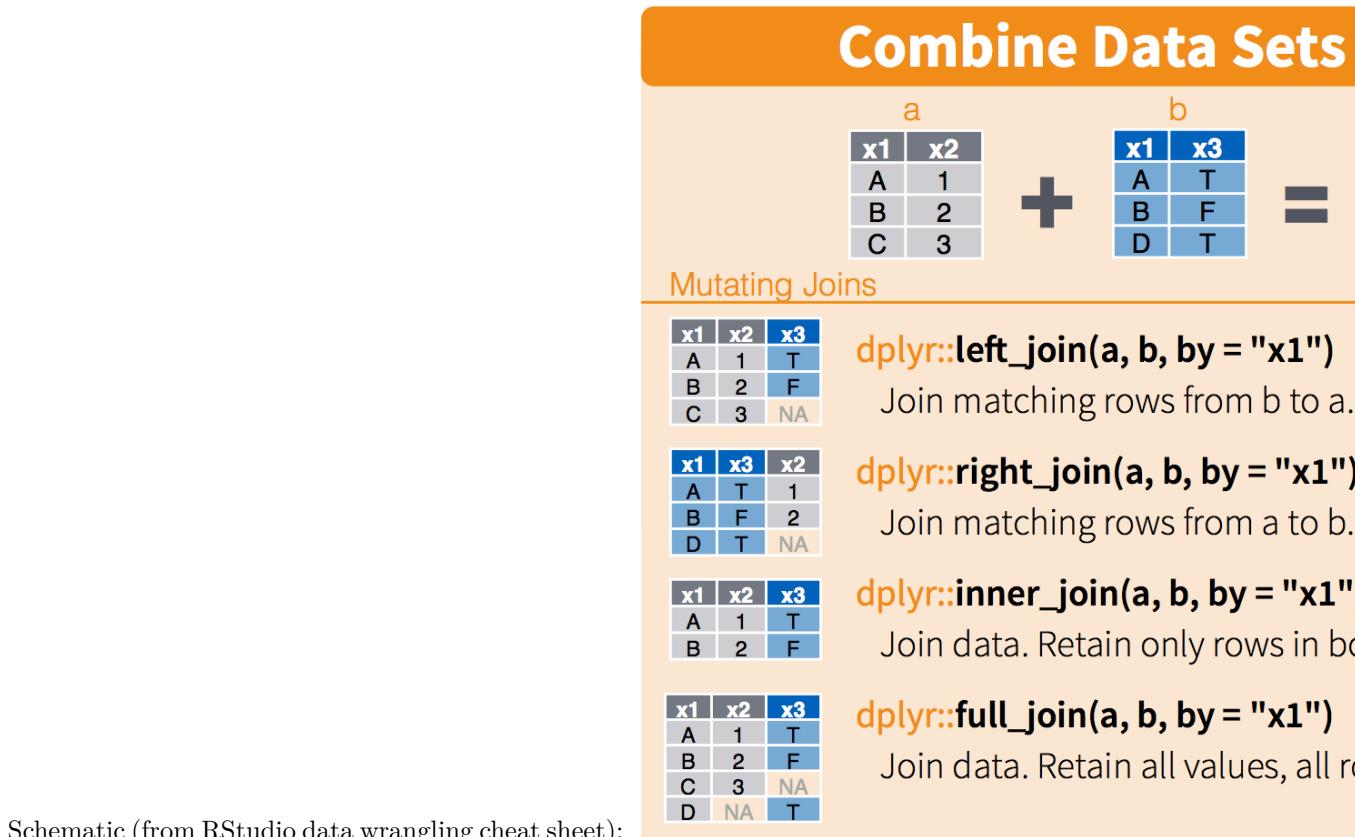
8.2.3 Merging data frames with `*_join()`

Excel's VLOOKUP can also be used to merge data from separate tables or worksheets. Here, we'll use the `*_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](#)):

- `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()`.
- `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.”
- `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.

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

Let's google this. That means copying this from the console and pasting it into Google.

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

Let’s 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. #### `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 `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*!

8.2.3.2 `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.

8.2.3.3 *_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") %>%
  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
```

8.2.3.3.1 Activity

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

Task: 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`)

Solution:

Sync your project with your repo on GitHub.

8.3 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 `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. `filter()` will return all observations (rows) that exactly match a condition.

8.4 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?)

8.5 Efficiency Tips

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

Chapter 9

readxl

9.1 Summary

9.1.1 Objectives

- create a new repo and give permission to a collaborator
- open as a new RStudio project, collaborate with a partner
- explore github.com blame, history, issues
- R communities and getting help (rOpenSci, RLadies)
- how to use Twitter for #rstats

9.1.2 Resources

9.2 Lesson

9.3 Efficiency Tips

9.4 Additional thoughts

Chapter 10

Synthesis

TODO: - stringr - Argaux, - Closing (15 mins)

10.1 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!

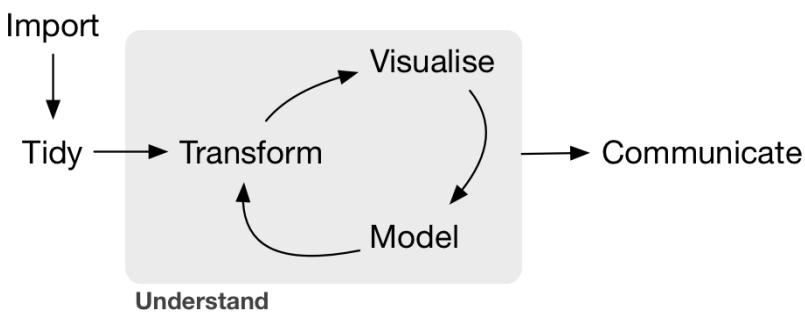


Figure 10.1: Grolemund & Wickham R4DS Illustration

10.2 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

10.3 Resources

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

10.4 Lesson

10.4.1 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`

10.4.2 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.

10.4.3 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()

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

10.6 Interludes (deep thoughts/openscapes)

10.7 Efficiency Tips