Case Studies in Reproducible Research: a spring seminar at UCSC

Eric C. Anderson, Kristen C. Ruegg, Tina Cheng, and the students of EEB 295

2017-06-02

Contents

Chapter 1

Course Overview

This is the home of the notes for a proposed course in data analysis and reproducible research using R, Rstudio, and GitHub.

The seminar is called, "Case Studies in Reproducible Research," but we utter that title with the caveat that, although the organizers have quite a few case studies they could spin up for this course, the case studies we will be studying in this course are going to be actual research projects that *you*—the participants—are working on. You're gonna bring 'em, and we are going to collectively help you wrassle them into a reasonable and reproducible data analysis. In the process we will touch on a number of elements of data analysis with R.

We will be working through a healthy chunk of the material in Garrett Grolemund and Hadley Wickham's book, R for Data Science, which is readable for free at the link above. We intend to use a handful of our own data sets each week to illustrate points from the book and show the material in action on real data sets.

This is not intended as a "first course in R". Students coming to the course should have at least a modicum of familiarity with using R, and we will launch more directly into using the tools of the tidyverse. EEB students with little or no experience in R might be interested in sitting in with Giacomo Bernardi's lab group on Mondays at 3PM in the COH library. They are conducting a Bio 295 seminar, working through "a super basic book that takes the very first steps into R."

For the interested, these materials were all prepared using RStudio's bookdown package. The RStudio project in which it lives is hosted on eriq's GitHub page here

1.1 Meeting Times, Location, Requirements

Intended to be Friday afternoons, 1:45–3:15 PM in the library/conference room at Long Marine Lab.

Students must bring a laptop to do examples during the seminar, and all students are expected to have a data set that they are in the midst of analyzing (or upon which they hope to commence analysis soon!) for a research project. We will

1.2 The origin of this seminar

The idea for this course was floated by Tina Cheng who was planning to lead a seminar in spring 2017 based in part on Eric C. Anderson's "Reproducible Research Course", taught at the Southwest Fisheries Science Center in the fall of 2014. Although going over those notes might have been a reasonable exercise, it turns out that a lot has changed in the world of data analysis since fall 2014, and the notes from that course are, today, a little bit dated.

We have been particularly excited by the ascendancy of Hadley Wickham's tidyverse approach to data analysis, and the tremendous development of a variety of tools developed by RStudio for integrating report generation and data analysis into reproducible workflows. In fact, Eric has been saying for the last year that if he were to teach another course on data analysis it would be structured completely differently than his "Reproducible Research Course". So, it was clearly time for him to stop talking and help put together an updated and different course.

At the same time, in working on our own projects and in helping others, we have consistently found that the most effective way for anyone to learn data analysis is to ensure that it is immediately relevant to whatever ongoing research project is currently consuming them. Therefore, in the current seminar, we are hoping to spend at least half of our time "workshopping" the data sets that seminar participants are actually involved in analyzing. Together we will help students wrestle their data, analyses, and ideas into a single, well-organized RStudio project under version control with git. Therefore, every student should come to this course with a data set and an associated analysis project.

1.3 Course Organizers

Kristen C. Ruegg Kristen is a conservation geneticist who specializes in the application of genome-wide data to understand population level processes and inform management, with a particular focus on migratory birds. She has has been enlightened to the powers of the "tidyverse" over the last couple of years (mostly through the constant insistence of her enthusiastic husband Eric Anderson) and is looking forward to becoming more fluid in its application over the course of the quarter. Her main role in this course will be to help with the course design and logistics and help reign Eric in when he has started to orbit into some obscure realm of statistical nuance.

Eric C. Anderson Eric trained as a statistician who specializes in genetic data. Since 2003 he has worked at the NMFS Southwest Fisheries Science Center in Santa Cruz. Although much of his statistical research involves the development of computationally intensive methods for specialized analyses of genetic data, he has been involved in a variety of data analysis projects at NMFS and with collaborators worldwide. Eric was an early adherent to reproducible research principles and continues, as such, performing most of his research and data analysis in the open and publicly available on GitHub (find his GitHub page here). In 2014, he taught the "Reproducible Research Course" at NMFS, and is excited to provide an updated version, focusing more, this time, on the recently developed "tidyverse".

Tina Cheng Tina is a graduate student in EEB. She is going to be leading the session during the first week of the course when Kristen and Eric are still on spring break, and then she is going to be joining in on the fun with us for the remainder of the quarter until she has to travel off to Baja, TA-ing the "supercourse" during the last four weeks of the quarter.

1.4 Course Goals

The goal of this course is for scientists, researchers, and students to learn to:

- properly store, manage, and distribute their data in a *tidy* format
- consolidate their digital research materials and analyses into well-organized RStudio projects.
- use the tools of the tidyverse to manipulate and analyze those data sets
- integrate data analysis with report generation and article preparation using the Rmarkdown format and using R Notebooks
- use git version control software and GitHub to effectively manage data and source code, collaborate efficiently with other researchers, and neatly package their research.

By the end of the course, the hope is that we will all have mastered strategies allowing us to use the above-listed, freely-available and open-source tools for conducting research in a reproducible fashion. The ideal we will be striving for is to be able to start from a raw data set and then write a computer program that

conducts all the cleaning, manipulation, and analysis of the data, and presentation of the results, in an automated fashion. Carrying out analysis and report-generation in this way carries a number of advantages to the researcher:

1. Newly-collected data can be integrated easily into your analysis.
2. If a mistake is found in one section of your analysis, it is not terribly onerous to correct it and then re-run all the downstream analyses.
3. Revising a manuscript to address referee comments can be done quickly.
4. Years after publication, the exact steps taken to analyze the data will still be available should anyone ask you how, exactly, you did an analysis!
5. If you have to conduct similar analyses and produce similar reports on a regular bias with new data each time, you might be able to do this readily by merely updating your data and then automatically producing the entire report.
6. If someone finds an error in your work, they can fix it and then easily show you exactly what they did to fix it.

Additionally, packaging one's research in a reproducible fashion is beneficial to the research community. Others that would like to confirm your results can do so easily. If someone has concerns about exactly how a particular analysis was carried out, they can find the precise details in the code that you wrote to do it. Someone wanting to apply your methods to their own data can easily do so, and, finally, if we are all transparent and open about the methods that we use, then everyone can learn more quickly from their colleagues.

In many fields today, publication of research requires the submission of the original data to a publicly-available data repository. Currently, several journals require that all analyses be packaged in a clear and transparent fashion for easy reproduction of the results, and I predict that trend will continue until most, if not all, journals will require that data analyses be available in easily reproduced formats. This course will help scientists prepare themselves for this eventuality. In the process, you will probably find that conducting your research in a reproducible fashion helps you work more efficiently (and perhaps even more enjoyably!)

1.5 Weekly Syllabus

1.5.1 Week 1 — Introduction and Getting Your Workspace Set Up

- At the end of this session we want to make sure that everyone has R, RStudio, and Git installed on their systems, and that they are working as expected.
- Additionally, everyoneshould have a free account on GitHub.
- And finally we need everyone's email address.

Some things to do:

- Get Rstudio cheat Sheets!
- Assemble data into a project
- Get private GitHub repos

Eric! You need to make an example project repo.

1.5.2 Week 2 — RStudio project organization; using git and GitHub; Quick RMarkdown

After this, students are going to have to put their own data into their own repositories and write a README.Rmd and make a README.md out of it.

1.5.3 Week 3 — Tibbles. Reading data in. Data rectangling

- Reading data into the data frames.
- read.table and read.csv
- tibbles
- The readr package
- Data types in the different columns and quick data sanity checks.
- A few different gotcha's
- Saving and reading data in R formats. `saveRDS` and `readRDS`.

1.5.4 Week 4 —

Chapter 2

Week One Meeting

Tina is going to be helping everyone get their systems all set up. After that we will have everyone clone an RStudio project from GitHub to see how easy that is.

2.1 Software Installation

1. **RStudio:** We want the latest "development" version of RStudio becuase it has features that we may want to use during this course. Get it from https://www.rstudio.com/products/rstudio/download/preview/ and install the appropriate one for your OS.
2. **R:** Let's make sure that we are all using the latest version of R. On March 7, 2017, version 3.3.3 was released. Go to https://cran.r-project.org/ and find the download link for your computer system. Download it and install it.
3. **bookdown:** This package is what I used to create these course notes. Getting it automatically installs a lot of other packages that are useful for authoring reproducible research. We want the latest development version, which can be obtained from GitHub by issuing the following commans at the R prompt (i.e. in the console window of RStudio:)

```
install.packages("devtools")
devtools::install_github("rstudio/bookdown")
```

4. Install **other packages** that we are going to be needing in the first few weeks. If you don't know how to install packages, ask Tina and she can show you. Install: `tidyverse`, and `stringr`.
5. Make sure that **git** is up and running on your system.
   - If you are using a Mac with a reasonably new OS, you should be able to just open the Terminal application (`/Applications/Utilities/Terminal`) and type "git" at the command line. If you have `git` it will say something that starts like:

```
    usage: git [--version] [--help] [-C <path>] [-c name=value]
   [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
   [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
   [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
   <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
```

```
clone      Clone a repository into a new directory
etc. etc. etc.
```

If you do not have `git` then it should pop up a little thing asking if you would like to install a reduced set of developer tools. You do. Click OK. **NOTE** Instead of a pop up it might say something like, "xcrun Error: invalid active developer path. etc. etc…". In that case, you can install a fresh set of command line tools by typing this at the command line:

```
xcode-select --install
```

- If you are using a PC, I can't be as much help, but you can find links with instructions on how to download `git` for a PC here.
- If you are using Linux then we will assume you know how to get `git` or that you already have it.

2.2 Get an account on GitHub

If you don't already have an account on GitHub, go to github.com and click the "sign up" link near upper right of the page. It is pretty self-explanatory. Go ahead and get a **free** account. There is nothing to pay for here!

2.2.1 Private repositories

If you are a graduate student and you do not feel comfortable posting your data on a public site like GitHub, then you should request some private repositories from GitHub. GitHub has a great deal for academic users like students: free private repositories. Please go to https://education.github.com/pack to sign up for your free student pack.

2.3 Open an RStudio Project from GitHub

I am going to have everyone use RStudio and GitHub to clone and open an RStudio project that I prepared as a template so that people can see how I would like them to start putting together their own projects.

To open this project, from RStudio, go to the menu option "File->New Project…". Then from the resulting dialog, choose "Version Control". Then choose "Git". Then it asks for a "repository URL". Supply this: `https://github.com/eriqande/rep-res-coho-example` and leave the "Project Directory Name" empty. And then choose a directory in which to put it and click OK.

Bam! That will pull the RStudio project off of GitHub, make a local clone of it on your hard drive and open.

Once you have done that. Open `README.Rmd` within the project, and click the "knit" button which should be present near the top left of the editor window.

That is how you convert an R Markdown README to README.md which is easy to read and see on GitHub.

If you want to see what the project repository looks like on GitHub, have a look at https://github.com/eriqande/rep-res-coho-example.

2.4 Assignment for next week: Create an RStudio Project with Your Own Data

Your mission for the following week—i.e., please have this done (or as done as you can get it) by Friday, April 14, 2017—is to prepare an RStudio project with your own data set, and provide some background about the data and the ways that you would like to analyze it. The "rep-res-coho-example" is an example of what I have in mind for this. You should use the README.Rmd from that project as a template for your own README.Rmd. (To do this you can just copy the README.Rmd file into the top level of your project directory and then edit it to reflect your own data and project.)

To do all this you are going to want to make your own project. Do that like this:

1. In RStudio, choose "File->New Project…"
2. Then choose "New Directory" and then choose "Empty Project"
3. In the next dialog, choose a name (*it is best to use only letters, numbers, dashes, and underscores, and include no spaces in the name*) for it **and be sure to click the "Create a git repository" button**.
4. Then click "Create Project".

That should give you a new project. Here are some guidelines for putting your own data in there

- Put all of your data in a directory named `data` in your project.
- CSV (comma separated values) is probably the best format to use. It is text-readable without proprietary software (unlike an Excel file); however if you need to look at it in a tabular way with Excel, (gasp), you can do that easily. Tab-delimited text works if you have that, but CSV is preferred.
- Use only letters, numbers, dashes, and underscores for the file names, (and periods for their extensions, i.e., `.csv`)
- Give a brief description of your data in the README.Rmd.

2.5 Reading for next week

This week (before Friday, April 14, 2017), please read the following sections of the R for Data Science book

- Workflow basics: super basic review on how R works.
- Workflow: projects: info about organizing RStudio projects.
- Workflow: scripts: how to evaluate code in scripts.
- tibbles: a streamlined data frame format.
- data import This is our key reading for the week.

When you are done with the *Data Import* reading, take a whack at writing some code to read the data files in your project into a variable (or several variables).

Chapter 3

Week Two Meeting

For this week, everyone should have completed the reading listed in Section **??**. And everyone should have at least been trying to set up an RStudio Project with their own data in it, and given a whack at reading their data into a variable.

3.1 Workflow and Project Recap

3.1.1 Workflow: basics

3.1.1.1 Style

I just want to reiterate a few things that might seem minor, but are stylistically important in the long run.

1. Don't use `=` for assignment. Use `<-`. On a Mac use Option-"-" to get that more quickly.
   - Hey! Note that you *do* use `=` (and **only** `=`) for *passing values to function arguments*.

```
# do this:
my_variable <- 10

# don't do this (it works but is not good style)
my_variable = 10


# do this
result <- my_function(arg1 = 10, arg2 = "foo")

# don't do this (it might return a results, but will likely be incorrect)
result <- my_function(arg1 <- 10, arg2 <- "foo")

# for example, figure out how this fails:
matrix(data <- 1:10, nrow <- 5, byrow <- TRUE)
```

2. Put spaces around both sides of `=`, and `<-`, and other mathematical operators like `+`, `-`, `*`, etc.
3. Put spaces after commas.
4. Use R-Studio's magical `Cmd-i` keyboard shortcut to automatically indent highlighted code.
5. If you want to really geek out, Hadley shares his style tips more completely in his Advanced R Programming book.
6. In fact, he now has updated those edicts for the tidyverse here.

This might seem pedantic, but adhering to these conventions makes it much easier for people to read your code.

3.1.1.2 TAB-completion

This is HUGE!! Start developing a twitchy left pinky now!

TAB early; TAB often!

Note that that completions of R code in the console or in the source window are context dependent:

- variables in global environment
- Functions
- Quoted strings complete to filenames in directories
- Installed packages in `library()`
- Help topics after `?`
- Function arguments within a function's parentheses. This is absolutely huge. If you can't remember all the different arguments a function takes, type the function and hit TAB within the parentheses. Try typing `read.table()` and his TAB while the cursor is inside the parentheses. Use the up and down arrows to scroll through the options, hit TAB again to insert one. Note that after you have used an argument it no longer appears in the list of options.

3.1.1.3 Ctrl + Shift + 1/2/3/4 to turn one of the panes to full-screen

Thanks to Diana for writing about practing that in her homework. Awesome feature that I've not used previously!

3.1.2 Workflow: projects

3.1.2.1 Disable saving of workspace for sure!

Let's all walk through this.

3.1.2.2 Another worthwhile preference for small-screens (like laptops)

Make sure that the RMarkdown preference is set to open in: **Window**. See Figure **??**.

3.1.2.3 Opening RStudio Projects from the OS (by clicking in the Finder)

- You can open an RStudio project by double clicking the RStudio Project icon from, for example, a Mac Finder window. It lives in a directory of the same name (but it has a `.Rproj` exension.)
- Or if you are a command line type, use, for example `open my_project.Rproj` from the Terminal.
- You can open as many RStudio projects as you like at a time.
- Each RStudio project launches its own, completely separate R session!
- Interestingly, if you click on the `.Rproj` file of a project that is open, RStudio will open another instance of that project. So, don't click on the `.Rproj` file for a project that is already open!
  - (In other applications on the Mac that will typically just take you to the currently open docuemnt, but not so with RStudio.)
- Use cmd-TAB to switch between open RStudio projects.

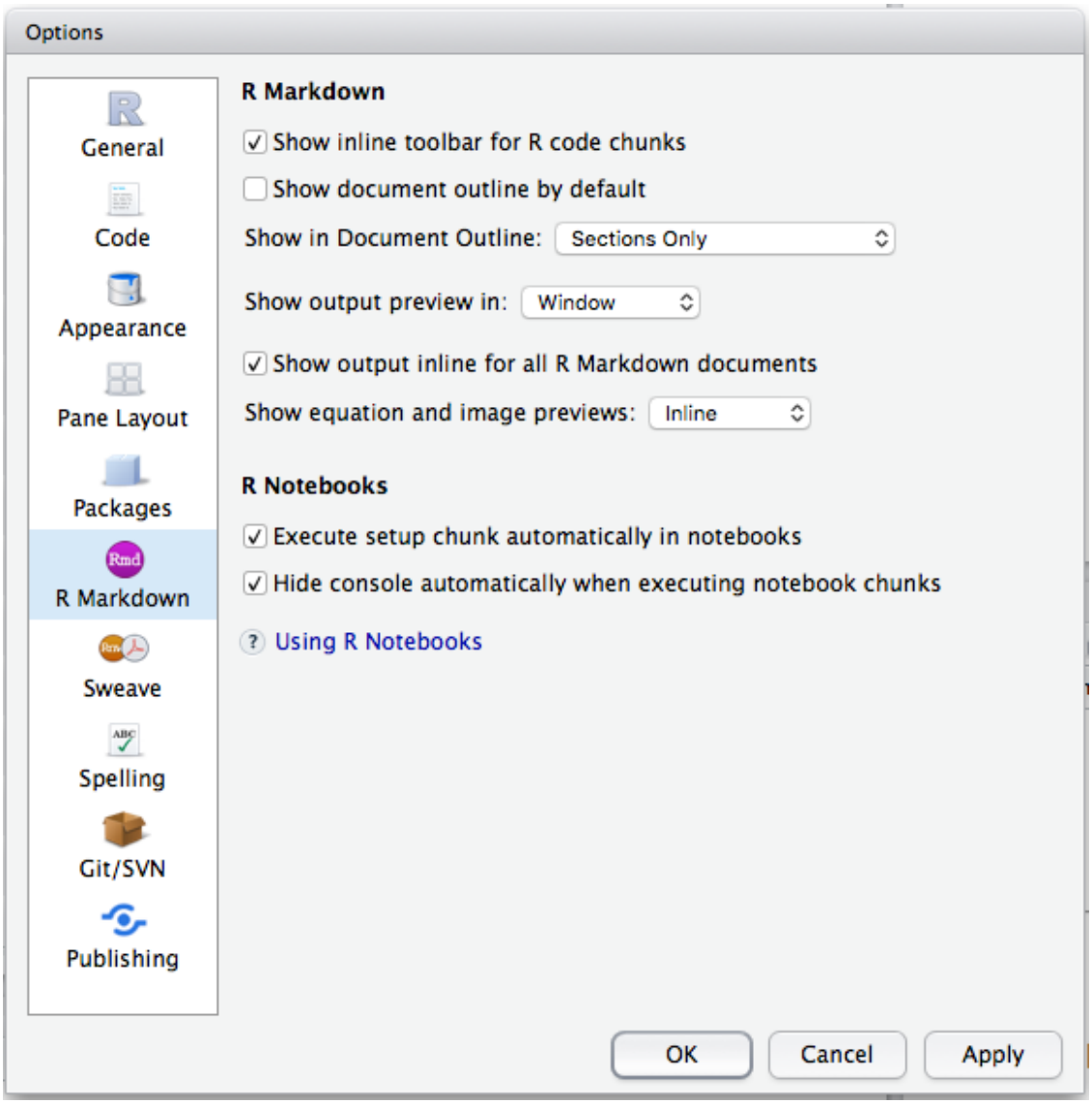


Figure 3.1: To read RMarkdown output in a separate page (highly recommended for laptops) choose "RMarkdown" on the left and choose "Window" from the dropdown menu, and click OK.

3.1.2.4 Opening RStudio Projects from RStudio

- When you open existing projects using the "File->Open Project…" menu option or with the "File -> Recent Projects" menu option and you currently have RStudio open "in another project,"" then the new project that you are opening jumps in "on top of" the previous one. It looks like your previous project has vanished into the ether. The OS thinks there is only one RStudio open, an it has the most recently opened project in it. WHERE'S MY OTHER ONE?!
- You can get back to it by clicking the project dropdown in the upper right of the project.
- However, if you switch between projects this way it restarts R each time you switch back to your project so it takes a lot of time and it is super-annoying.
- If you are working concurrently in multiple projects, I recommend opening them from the Finder (or Terminal) and switching between them using **Cmd-TAB**.

3.1.2.5 What is the `.Rproj` file, really

It is just a text file that stores some information and any project-specific preferences if there are any. Here is what `rep-res-eeb-2017.Rproj` looks like if you open it with a text editor:

```
Version: 1.0

RestoreWorkspace: Default
SaveWorkspace: Default
AlwaysSaveHistory: Default

EnableCodeIndexing: Yes
UseSpacesForTab: Yes
NumSpacesForTab: 2
Encoding: UTF-8

RnwWeave: knitr
LaTeX: pdfLaTeX

AutoAppendNewline: Yes
StripTrailingWhitespace: Yes

BuildType: Website
```

3.1.2.6 R in an RStudio project launches in the project directory

- This makes reproducibility much easier. You can find and load files using *relative* paths.

- Everything you might be accessing from R (data, scripts, etc.) or outputting from R will be easy to get to if they are "in the project"
- When we say that a file is "in the project" we mean that it is stored on disk somewhere within the project directory.
- The project directory (sometimes called the *root* of the project directory) is just the directory that contains the `.Rproj` file.
- Expert user tip: `rprojroot::find_rstudio_root_file()` (part of the `rprojroot` package) let's you find the root of an RStudio project directory. This can be helpful sometimes….

3.1.3 Workflow: scripts

- Script editor window vs console window

- Keyboard shortcuts for evaluating codes in your scripts:
  - **Cmd-Return** (sends current line to console and advances cursor to next line)
  - **Highlight with Cmd-Return** (send highlighted code to console)
    * For this, **Shift-up-arrow** and **Shift-down-arrow** are good for highlighting.
    * As is **Shift-Command-right-arrow** or **Shift-command-left-arrow**.

3.2 Let's talk about the pipe %>%

For anyone who had ever worked comfortably in Unix for a long time, and was used to chaining the output of one utility in as the input for another utility using the pipe: `|`, R's syntax for composition of functions was always super cumbersome and required all sorts of nasty, nested parentheses.

Consider this simple set of operations: imagine we want to

1. simulate 1000 gamma random variables, G, with parameters $\alpha = 5$ and $\beta = 1$,
2. for each G simulate a Poisson random variable with mean (`lambda`) G.
3. take the `sqrt` of each such variable
4. compute the variance of the result

This can all be done in one line, but is ugly!

```
# set random seed for reproducibility
set.seed(5)

var(sqrt(rpois(n = 1000, lambda = rgamma(n = 1000, shape = 5, scale = 1))))
```

```
## [1] 0.5828768
```

It doesn't matter how stylishly you include spaces in your code, this is just Fugly!

You can write it on multiple lines, but it is friggin' ghastly! Maybe worse than before.

```
set.seed(5)

var(
  sqrt(
    rpois(n = 1000, lambda = rgamma(
      n = 1000, shape = 5, scale = 1
    )
    )
  )
)
```

```
## [1] 0.5828768
```

The problem is that the order in which the operations are done does not match the way things are written: the first thing to get done is the call to `rgamma`, which is nested deeply within the parentheses.

Enter the R "pipe" symbol. It is not as convenient to type as `|`, but you can make it quickly with the keyboard shortcut `cmd-shift-M`: `%>%`. This was introduced in the `magrittr` package, and the `tidyverse` imports the `%>%` symbol from `magrittr`.

Behold!

```
library(tidyverse)
set.seed(5)

rgamma(n = 1000, shape = 5, scale = 1) %>%
  rpois(n = 1000, lambda = .) %>%          # pass G is as the lambda parameter using the dot: .
```

```
  sqrt() %>%                              # no dot here, so the previous result is just the first argu
  var()                                   # same here
```

```
## [1] 0.5828768
```

That is a hell of a lot easier to read! It gives me goose bumps it is so elegant.

The %>% symbol says, "take the result that occurred before the %>% and pass it in as the . in whatever follows the %>%." Furthermore, if there is no . in the expression after the %>%, simply pass the result that occurred before the %>% in as the *first argument* in the function call that comes after the %>%.

This type of "chaining" of operations is particularly powerful when operating on `tibbles` using `dplyr`

3.3 Tibbles and "rectangular" data

- gonna talk a little about data types too.
- Chinook CWT data example.
- Get comfy with the `View()` function!

3.3.1 Tibble excercises

I'm gonna just blast through these here in case people are curious. These are answers I would use.

1. Use `class()`

   ```
   class(mtcars)
   ```

   ```
   ## [1] "data.frame"
   ```

   ```
   class(tibble::as_tibble(mtcars))
   ```

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

   Hey! does everyone see the `tibble::as_tibble()` there? The `::` is the "namespace addresser". It lets you run a function from a library without loading the library. If you already have done `library(tidyverse)` you would have loaded the `tibble` library and could just write `as_tibble(mtcars)` but I wanted to be explicit about where the `as_tibble()` function comes from. (As an aside, it turns out that this is how you would write it if you were writing code for a package.)

2. Let's do it first as a `data.frame`:

   ```
   library(tidyverse)
   df <- data.frame(abc = 1, xyz = "a")
   df$x
   ```

   ```
   ## [1] a
   ## Levels: a
   ```

   ```
   df[, "xyz"]
   ```

   ```
   ## [1] a
   ## Levels: a
   ```

   ```
   df[, c("abc", "xyz")]
   ```

   ```
   ##   abc xyz
   ## 1   1   a
   ```

   And then we can do it again as a `tibble`

```
df <- tibble(abc = 1, xyz = "a") %>%
  as_tibble()
df$x
```

```
## Warning: Unknown column 'x'
```

```
## NULL
```

```
df$xyz
```

```
## [1] "a"
```

```
df[, "xyz"]
```

```
## # A tibble: 1 × 1
##     xyz
##   <chr>
## 1     a
```

```
df[, c("abc", "xyz")]
```

```
## # A tibble: 1 × 2
##     abc   xyz
##   <dbl> <chr>
## 1     1     a
```

Aha! Things to notice are:

a. `data.frame()` coerces to factors.
b. tibble doesn't do partial name matching `$x`≠`$xyz`
c. Square bracket extraction of a single column of a tibble retains its tibbleness. Not so with `data.frame`. With `data.frame` it gets turned into a vector.
d. `$` extraction with `tibble` returns a vector.

3. Let's make `mtcars` a `tibble`

```
mtcars_t <- as_tibble(mtcars)
var <- "mpg"

# this will get the "mpg" column out but retain it as a tibble
mtcars_t[, var]
```

```
## # A tibble: 32 × 1
##      mpg
##    <dbl>
## 1   21.0
## 2   21.0
## 3   22.8
## 4   21.4
## 5   18.7
## 6   18.1
## 7   14.3
## 8   24.4
## 9   22.8
## 10  19.2
## # ... with 22 more rows
```

```
# and this will just grab the column as a vector
mtcars_t[[var]]
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

4. Remember that non-syntactic names (those that do not start with a letter or underscore and which include characters other than - and "_" and ".") must be enclosed in backticks. Let's get our data:

```
annoying <- tibble(
`1` = 1:10,
`2` = `1` * 2 + rnorm(length(`1`))
)
```

OK, now let's do the questions:

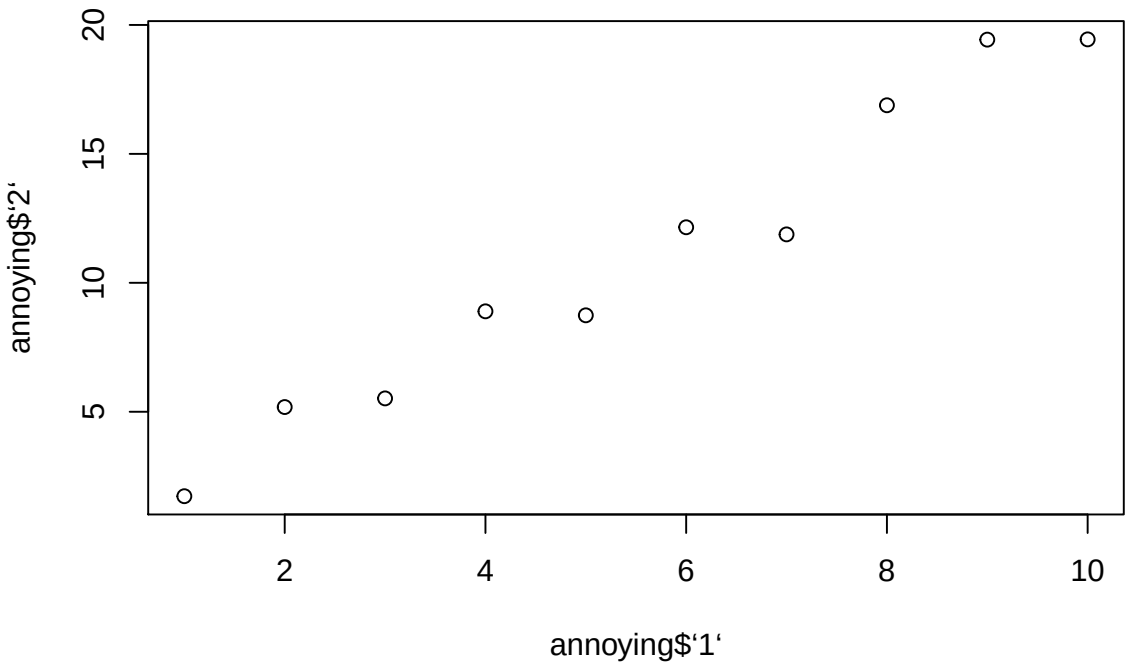
1. Use dollar sign with backticks:

```
annoying$`1`
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

2. Use dollar sign with backticks

```
plot(annoying$`1`, annoying$`2`)
```



3. Use `mutate` (we haven't talked about this yet) with backticks

```
annoying %>%
  mutate(`3` = `2` / `1`)
```

```
## # A tibble: 10 × 3
##      `1`      `2`      `3`
##    <int>    <dbl>    <dbl>
## 1      1 1.725450 1.725450
## 2      2 5.182598 2.591299
## 3      3 5.517888 1.839296
## 4      4 8.894382 2.223596
## 5      5 8.740021 1.748004
## 6      6 12.153600 2.025600
```

```
## 7      7 11.878134 1.696876
## 8      8 16.885478 2.110685
## 9      9 19.429848 2.158872
## 10    10 19.439987 1.943999
```

4. Use `rename` (we haven't talked about this yet) with backticks

```
annoying %>%
  mutate(`3` = `2` / `1`) %>%
  rename(one = `1`,
         two = `2`,
         three = `3`)
```

```
## # A tibble: 10 × 3
##      one       two    three
##    <int>     <dbl>    <dbl>
## 1      1  1.725450 1.725450
## 2      2  5.182598 2.591299
## 3      3  5.517888 1.839296
## 4      4  8.894382 2.223596
## 5      5  8.740021 1.748004
## 6      6 12.153600 2.025600
## 7      7 11.878134 1.696876
## 8      8 16.885478 2.110685
## 9      9 19.429848 2.158872
## 10    10 19.439987 1.943999
```

5. Look it up with `?enframe`. It turns out that `enframe()` is super useful.
   Often you will have a vector of values with names associated with it. For example:

```
v <- c(1, 3, 4, 10)
names(v) <- c("a", "b", "b", "c")
v
```

```
##  a  b  b  c
##  1  3  4 10
```

   If you want to deal with this type of vector in the tidyverse, you can enframe it into a tibble:

```
enframe(v)
```

```
## # A tibble: 4 × 2
##    name value
##   <chr> <dbl>
## 1     a     1
## 2     b     3
## 3     b     4
## 4     c    10
```

   By default it makes columns of "name" and "value". That is awesome!

6. Do `package?tibble` and read through it to find that the answer we want is `tibble.max_extra_cols`.

3.4 Data import

- Why use `readr` instead of the base-R reading functions? Plenty of reasons.
- Explicit column specifications if you want to do that.

3.4.1 RStudio's GUI importer

- This is a great way to start importing CSV files.
- Don't do it every time! use the code that it creates to make reading your data in reproducible.

Chapter 4

Week Three Meeting

4.1 Git Basics

Goals for Today:

- Explain what git is (and how it is different than GitHub)
- Introduce the sha-1 hash (for fun!)
- Get familiar with RStudio's very convenient interface to git
  - staging files
  - unstaging file
  - viewing differences between staged and unstaged files
  - committing files
  - viewing the commit history

4.1.1 An overview of Version Control Systems (VCS)

- Git is a type of VCS
- At its crudest, a VCS is a system that provides a way of saving and restoring earlier versions of a file.

4.1.1.1 A typical VCS for a non-computer programmer

- Start writing `my_manuscript.doc`.
- At some point worry that MS Word is going to eat your file, so,
  - Make a "backup" called `my_manuscript_A.doc`
- Then, before overhauling the discussion, save the current file as `my_manuscript_B.doc`.
- Email it to your coauthors and then have a series of files with other extensions such as the initials of their names when they edit them and send them back.
- Etc.
- Disadvantages:
  - Hard to find a good record of what is in each version. (Wait! I liked the introduction I wrote three weeks ago…where is that now?)
  - A terrible system if you have multiple files that are dependent on one another (for example, figures in your document, or scripts and data sets if you have a programming project.)
  - If you decide that you want to merge the changes you made to the discussion in version `_C` with the edits on the introduction in version `_K`, it is hard.

4.1.1.2 Other popular VCS systems

- rcs, cvs, subversion, etc.
- These all had a "Centralized" model:
  - You set up a repository on a server that has the full version history,
  - then each person working on it gets a copy of the current version, and nothing more.
  - They can submit changes back to the central repository which tries to deal with conflicting submissions.
  - You need to be online to do most operations.
- I used a few of these, and missed them for a few weeks when I switched to Git, but then never looked back and couldn't imagine using them again.

4.1.1.3 The Git model — Distributed Version Control

- Git stores "snapshots" of your collection of files in a repository
- For our work, the "collection of files" will be "the stuff in your RStudio project"
  - Another reason it is nice to keep everything you need for a project together in a "project directory"
  - Though git scans your project directory for new additions and changes, it will not add a new file, or add new changes, to the repository until you *stage* and subsequently *commit* the file.
- When you clone a repository, **you** get the whole version history
- When someone else clones that repository, **they also** get the whole version history.
- Git has well-developed features for merging changes made in different repositories
  - But, for today, we will talk mostly of a single user interacting with git.

4.1.2 Git versus GitHub

- They are not the same thing!
- Git is software that you can run on your own machine for doing version control on a repository.
  - It can be *entirely* local. i.e. only on your hard drive and nowhere else.
  - This is super-useful for any project, because solid version control is great to have.
- GitHub is a website, with tools powered by Git (and many that they brewed up themselves) that makes it very, very easy to share git repositories with people all over the place.

4.1.2.1 Is everything on GitHub public?

- No! Many companies use GitHub to host their proprietary code
  - They just have to pay for that…
- By default, you can put anything on GitHub for free as long as it is under a fairly free-use (open source type) license and it is available to anyone
- If you want a private repository, as an academic affiliate you just have to ask and GitHub will give you unlimited private repos for free.
- And if you are a student and you have not yet done so, go to https://education.github.com/pack to sign up for your free student pack.

4.1.3 Using git through RStudio

- Now we can do a few things together to see how this works.
- Most of the action is in the Git Pane…
- Today we will talk about:
  - Staging files (preparing them to be **committed**)
  - Committing files (putting them into the repository)
  - viewing differences between staged and unstaged files

  - committing files
  - viewing the commit history

4.1.3.1 Two final configurations before starting:

- Open the shell (Tools->Shell…) and issue these two commands, replacing the name "John Doe" with yours, and his email with yours.
  - You may as well use the email address that you gave to GitHub, though it doesn't necessarily have to be

    ```
    git config --global user.name "John Doe"
    git config --global user.email johndoe@example.com
    ```

    You only need to do this once.
- Finally, if you are using a Mac, configure it to cache your GitHub credentials so you needn't give your password every time you push to it:

  ```
  git config --global credential.helper osxkeychain
  ```

4.1.3.2 The status/staging panel

- RStudio keeps git constantly scanning the project directory to find any files that have changed or which are new.
- By clicking a file's little "check-box" you can stage it.

- Some symbols:
  - **Blue-M**: a file that is already under version control that has been modified.
  - **Yellow-?**: a file that is not under version control (yet…)
  - **Green-A**: a file that was not under version control, but which has been staged to be committed.
  - **Red-D**: a file under version control has been deleted. To make it really disappear, you have to stage its disappearance and commit. Note that it still lives on, but you have to dig back into your history to find it.
  - **Purple-R** a file that was renamed. (Note that git in Rstudio seems to be figuring this out on its own.)

4.1.3.3 The Diff window

- Shows what has changed between the last committed version of a file and its current state.
- Holy smokes this is convenient
- (Note: all this output is available from the command line, but the Rstudio interface is very nice, IMHO)

4.1.3.4 Making a Commit

- Super easy:
  - After staging the files you want to commit…
  - Write a brief message (first line short, then as much after that as you want) and hit the commit button.

4.1.3.5 The History window

- Easy inspection of past commits.
- See what changes were made at each commit.

4.1.4 Go for it everyone!

- Make some changes and commit them yourselves.

- Add some new files to the project, and commit those.
- Get familiar with the diff window.
- Check the history after a few commits.

4.1.5 How does git store and keep track of things

- Everything is stored in the .git folder inside the RStudio project.
- The "working copy" gets checkout out of there
- Committed changes are recorded to the directory

4.1.5.1 What is inside of the .git directory?

We can use R to list the files. My `rep-res-course` repository that hold all the materials for a course like this one looks like:

```
## check out this file-system command in R
dir(path = ".git", all.files = TRUE, recursive = TRUE)
```

The output from that command looks something like this:

```
  [1] "#MERGE_MSG#"                                   "COMMIT_EDITMSG"
  [3] "COMMIT_EDITMSG~"                               "config"
  [5] "description"                                   "FETCH_HEAD"
  [7] "HEAD"                                          "hooks/applypatch-msg.sample"
  [9] "hooks/commit-msg.sample"                       "hooks/post-update.sample"
 [11] "hooks/pre-applypatch.sample"                   "hooks/pre-commit.sample"
 [13] "hooks/pre-push.sample"                         "hooks/pre-rebase.sample"
 [15] "hooks/prepare-commit-msg.sample"               "hooks/update.sample"
 [17] "index"                                         "info/exclude"
 [19] "logs/HEAD"                                     "logs/refs/heads/gh-pages"
 [21] "logs/refs/heads/master"                      "logs/refs/remotes/origin/gh-pages"
 [23] "logs/refs/remotes/origin/master"           "objects/00/906f99e192ff64b4e9e9a0e5745b0a4f841cbd"
 [25] "objects/01/ab18d4ce04fb06532bb06ed579218fef89d478" "objects/02/74554e0b574b9beb2144f26ad392583005
 [27] "objects/03/2d224bf78798e8b9765af6d8768ade14694a9d" "objects/04/03d552ab37b0bcaeebed0ac3068d669261
 [29] "objects/04/4a12f8ccc12a4a5ba84ab2bf5a1ae751feea6f" "objects/04/9ec3065bb0434ded671fa83af5ade803bc
 [31] "objects/04/ea8efb1367727b081dea87e63818be0a4d02f0" "objects/05/b22ecc373d5058e36d7ca773a4475c46da
 [33] "objects/07/8831b46c9b63e8c2d50b79304ed05de9274c28" "objects/07/b57af2a0cbd0545a6cd3e93f10cc5d768e
 [35] "objects/08/674e6e4d534b3424e2629510d20bb6d1b0be94" "objects/08/8b282d5b978dc1ff6eef3871d3fb3a9256
 [37] "objects/09/565dcd10d7adc0551783b443e8fd71486b3997" "objects/09/6828a0cfb96f30d6e99cfc04a5c1686b9e
 [39] "objects/0a/30fe678abc342c58daab0ad42163b371babda0" "objects/0a/b71109dae6e5711755feddfb06b81b1376
 [41] "objects/0b/442fdfc183783537985c17151ae3483fa00cf6" "objects/0b/c0451fd0e7081a7db05fdd38b12870bdca
 [43] "objects/0c/0f7cf8c73d901795dad4bd5f504c53c3bf2093" "objects/0d/14a7a2a19ffc3b9820f011e3270c965a5f
 [45] "objects/0e/35cfb4d55e52d27083b8d2eccab9296b920d76" "objects/0e/7bba5882077a8b00a76d3eabe6b23cadc6
 [47] "objects/0e/8abf4cc0885a727ee2459fdbb272828e267cc4" "objects/0f/1f3f7be7787d5d44dc1155f3b7a44eddc9
 [49] "objects/10/54d2e7a9baf61618521c522b15db40855b3431" "objects/11/7d874e1616500b5fba51b9f0ee1e8d0fbe
 [51] "objects/11/c33cc1d5c8de7c7cbf7257b7d32f7ca3d458ef" "objects/14/1ccea514e106e20eef47b791a23e036d1f
 [53] "objects/15/cc3a6f15dadb3446ad0af34a3ecde8d81d65f9" "objects/15/ddaf45bf00c3ef2d8f499ebd6dc3a86bf9
 [55] "objects/16/0c9386dfa9707d81fbbbcc52f0c7638703f9a9" "objects/16/8ee93b6a4612dbd76bc06a49460df9f9f6
```

Yikes!

4.1.5.2 How does git know a file has changed?

- Does it just look at the modification date?
- NO! It "fingerprints" every file, so it knows when it has changed from the most recent committed version.
  - Demonstration. Change a file. Save, then undo the change and save again…Git knows the file has been changed back to its "former self"
- SHA-1 hashes. We will learn more about those later.

- You will see things like `ed00c10ae6cf7bcc35d335d2edad7e71bc0f6770` all over in Git-land.
- You can treat them as very specific names for different commits.

4.1.5.3 What should I keep under version control?

- General rule: don't keep derived products.
  - i.e. If you have an Rmd file that creates an html file, there isn't much need to put the html file under version control with git, because you can just regenerate it by Knitting the Rmd file.
- Do keep data, source code, etc.
- Sometimes certain outputs and intermediate results from long calculations can be committed so that you don't have to run a 4 hour analysis to start where you were before.
- For such results, consider `saveRDS()` with the `compress = "xz"` option (and its companion `readRDS()`.

4.1.5.4 How can I make git ignore certain files?

- The `.gitignore` file!
- File names (and patterns) in the `.gitignore` file are ignored *recursively* (down into subdirectories), by default.
- Files won't be ignored if they are already in the repository.
- Example: `*.html`

4.2 Pushing and Pulling With GitHub

4.2.1 Creating a Repository on GitHub and the initial push

When you have an RStudio project under git version control on your laptop or desktop computer, creating a remote repository on GitHub is quite easy. A few steps:

1. Upper right corner: "create new" button (a "+" with a little triangle.) Choose "New Repository"
2. Give it a name. It makes most sense to name it the same as the RStudio project you want to push up there. So, for example, if my project file was `boing.Rproj`, I would name the repository `boing`.
3. Add a 5 or 6 word description if you want.
4. Choose **public** or **private**
5. DO **NOT** choose to "Initialize this repository with a README". You likely already have a README. Initializing the repository with one will create headaches.
6. Also, don't add a `.gitignore` or a license (select "none", which should be the default, for both of those)
7. Click the green "Create Repository" button

That will take you to another screen. In the middle find the code box below the heading, **…or push an existing repository from the command line**.

1. Copy that two lines of code from you web browser. It will look something like this:

```
git remote add origin https://github.com/eriqande/boing.git
git push -u origin master
```

but it will be specific to the repository you just made, so the URL and name of the repo will be different than what you see above. Note, you can copy the lines by clicking the "copy this text" icon on the right side of the page.

2. Go to RStudio, in the project that you want to push to GitHub, and choose "Tools-Shell". That will give you a terminal window. Paste the commands you copied into that terminal window and hit return.
3. It might ask you for your GitHub username and password.

Voila!

4.2.2 Subsequent pushes

Once you have pushed the repo up there. Try making some changes on your laptop, committing them, and then hitting the "Push" button on the git panel…

4.2.3 Assign collaborators

From the repository page on GitHub, choose "Settings" (on the upper right) and find the "collaborators" link (on the left).

If you have a private repository, you can add GitHub user `eriqande` (that's me…) to it and I will be able to view it and give comments and suggestions.

4.3 Next Week's Assignment

- With luck, we will get everyone's projects up to GitHub before the end of our session today. However, if we don't, please get that done ASAP.
- The big assignment is to read the Data Transformation chapter in "R for Data Science." *Warning*: This is a long and meaty chapter, so get an early start! The chapter goes through what you need to know to leverage all the `dplyr` goodness. Please work all the examples, and do the exercises. In fact, when you are working through the examples with the `nycflights` data set, you should, after each example, try to do the same type of operation on your own data set.

Chapter 5

Week Four Meeting

This was a bit of a free-form discussion on a variety of topics.

5.1 Knit your README.Rmd files

First thing we talked about was the fact that GitHub will render a `README.md` file to an html web page that is nice and easy to read. It will sort of render a README.Rmd file, but it won't do everything to it. Namely:

1. The YAML header block comes out as a table.
2. It will **not** evaluate all the R code and deliver the results.

Rather, it is necessary to locally *knit* the `README.Rmd` file to create a `README.md`, then this `README.md` file must be committed to the repository and pushed. This needs to happen each time you have updated the `README.Rmd` file.

Note that your `README.Rmd` file should start with the following:

```
cat(readLines("inputs/readme-header.txt"), sep = '\n')
```

````
---
title: "NameOfPackage"
date: "`r format(Sys.time(), '%d %B, %Y')`"
output:
  github_document:
    toc: true
---

<!-- README.md is generated from README.Rmd. Please edit that file -->

```{r, echo = FALSE}
knitr::opts_chunk$set(
  collapse = TRUE,
  comment = "#>",
  fig.path = "readme-figs/"
)
```

Then start adding your text here...
````

5.2 Changing to factors

Mikki had a question: she wanted to have a column that contained 1's and 2's as factors. her data set had several entries that were "1,2". She wanted to convert those to 2's and then make them all factors. We discussed how this could be done with dplyr. The important message was that dplyr does not change the original input variable, but in the output, you can "mutate over the top of an existing variable" (i.e. in the output the column will have been changed, but not in the original input data frame).

5.3 The `group_by()` function

We spent a bit of time going over how to think about what the `group_by()` function does. Eric likes to think of it as breaking your original tibble up into a lot of different tibbles, according to the grouping variables, after which, each little tibble gets sent to the following verb (`summarise()`, `mutate()`, `filter()`, etc.)

We talked about that fact that while it is quite natural to think about using the `group_by()` function in conjunction with `summarise()`, it is also very powerful to be able to use it in conjuntion with `mutate()`.

When you do a summarise, only the grouping variables and the newly-created summary variables get returned in the output tibble, and the rows are arranged by the grouping variables. When you do a `group_by()` and then `mutate()` all of the columns get returned and there is no automatic arranging that goes on.

5.4 How do I learn about all the vectorized functions I can use in `mutate()` and `summarize()`?

There was consensus in the class that even once we have learned the mechanics of using `mutate()` and `summarise()`, we might still be at a loss as to *how* to use them, or with which functions. Admittedly, there are many, many vectorized functions in R that you might apply within a `mutate()` or `summarise()` function, and learning about all of those, and having them at your fingertips when you need them is part of the never-ending journey of gaining experience with R.

However, there are a few things that can help with that journey. Here are my two favorite suggestions:

1. Get the RStudio *dplyr cheatsheet*. In RStudio, Go to Help–>Cheatsheets–>Data Manipulation-with-dplyr,-tidyr. While you are at it. Check out their other cheatsheets.
2. Review Hadley's Advanced R recommended vocabulary.
   This is a nice, compact list of R functions that you should be familiar with, or at least aware of.

5.5 Next Week's Assignment

For Week 5, we are going to talk about **joins**. This is a very important topic for combining data from different data sets. Thus, everyone should read Chapter 13: Relational Data in the R for Data Science Book. This is an amazing chapter, and will go a long way in helping people understand how to make their lives easier when it comes to combining multiple tibbles of information (for example, a tibble of metadata for each individual and a tibble of genotype information, for the same individauls, etc.). Try to work through all the examples and do the exercises.

Chapter 6

Week Five Meeting

6.1 Some things regarding people's repositories

6.1.1 Data Compression

We have been endorsing `.csv` as a good format for data, and *it is*, because it is human-readable and easily parsed into tibbles. However, when you have very long tibbles, it is not necessarily the most space-efficient format. Large `.csv` files can take up much more space on your hard drive *than they should*.

What do we mean by "*than they should?*" in that context? This has to do with how much *information* is in the file, where information is used in the context of *information theory*. Often tibbles will have columns that have fairly "redundant" information—for example, in a mutlispecies salmon data set, one column might have entries that are either "Chinook", "coho", or "steelhead". It takes a few bytes to store each one of those words, and if they are used in a column that has millions of rows, that can add up to a lot of space on your hard drive. Colloquially, *data compression* is the art of finding ways of using short "code-names" for things or patterns that occur frequently in a file, and in so doing, reducing the overall file size.

The consequences of data compression can be profound and wonderful. You can reduce the size of a file, sometimes by an order or magnitude or more. There are a few good choices available for compressing your data (making it smaller.). Note that doing so often makes it a little harder to edit your data set; however, if your data set is not going to change, and it is large, then it makes sense to compress it—especially if it is so big you would rather not (or can't) put it on GitHub.

6.1.1.1 gzip

If you are working on a Mac, you have the Unix utility `gzip`. We will illustrate its use on Katie's big salmon data set, `ASL_merged.csv`. Let's see how big that is. We can use the Unix utility `du` (stands for "disk usage").

```
# give this command on the Terminal in the directory where the file lives:
du -h ASL_merged.csv
```

The results comes back:

```
322M    ASL_merged.csv
```

Whoa! This file is 332 Megabytes. That is quite large!

However, we can compress it like this:

```
gzip ASL_merged.csv
```

When we do that, it compresses the file and renames it to have a `.gz` extension on it: `ASL_merged.csv.gz`. We can then see how big that file is:

```
du -h ASL_merged.csv.gz
```

tells us:

```
11M ASL_merged.csv.gz
```

Whoa! We went from 332 Megabytes to 11. It is just 3% of its original size (and small enough that you can safely put it on GitHub).

One very nice feature is that gzipped files can be read in directly by the functions of the `readr` package. So, for example, `read_csv()` works just fine on the gzipped version of Katie's massive salmon data set:

```
# this works the same as it would on the ungzipped file
salmon <- read_csv("data/ASL_merged.csv.gz")
```

6.1.1.2 xz comression with saveRDS()

Another method that can be even more efficient with tibbles is to store them as R objects using the `saveRDS()` function with the `xz` compression option. This has the nice advantage that all the data types of the variables (for example, if you had made factors out of some) will be preserved *exactly* as they are in the tibble when you save it.

Let's imagine we have read the tibble into the variable `salmon`, and all the column types were as we wanted. Then, we could save that tibble directly to a compressed file like this:

```
saveRDS(salmon, file = "ASL_xz.rds", compress = "xz")
```

Note that `compress = "xz"` option. Let's see how that did using `du` on the Unix terminal:

```
du -h ASL_xz.rds
```

tells us:

```
3.7M    ASL_xz.rds
```

Holy Smokes! Only 3.7 Megabytes. That is only 1.1% of its original size. Lovely!

In order to read that tibble back into a variable (named `my_var`, say) in R, you would use `readRDS()` like this:

```
my_var <- readRDS(file = "ASL_xz.rds")
```

Voila!

6.2 A quick aside about missing data

Garrett and Hadley note that "missing values are 'contagious': almost any operation involving an unknown value will also be unknown." This is true for the most part, but there is a vexing inconsistency. Observe

This gives us NA as we would hope it would

```
NA == 0 | NA == 1
```

```
## [1] NA
```

However, this one returns FALSE. What gives?

```
NA %in% c(0, 1)
```

```
## [1] FALSE
```

6.3 Brief Highlights of the Joins Chapter

You will likely end up using joins all the time. As noted in the book, the `left_join()` is what you will likely use all the time. In this case you have a "focal" data frame with all the rows ("cases" or "observations") in the `x` table that you are going to be wanting to add some columns to. Those columns live in the `y` table (along with the matching keys).

"*The left join should be your default join: use it unless you have a strong reason to prefer one of the others.*"

You an do the same things with base R's `merge()` function, but it is slower and somewhat harder to express your intent with it. (I've always really disliked the `merge()` function…)

6.3.1 A few thoughts on keys

It is always a worthwhile exercise to go through and figure out what the *primary key* is in a tibble you are working with. It might be that the primary key is a compound key: it defines unique observations by a combination of several variables. Sometimes there is no explicit primary key! It is worthwhile to add a *surrogate key* in that case.

On the flip side of these issues: when you are compiling your own data set, you might want to spend some time making sure that units that might be relevant to an analysis are explicitly identified in a single column. Here is an example: the NOAA Observer program takes tissue samples from bycatch for genetic analysis. There is a primary key `tissue_sample` for every tissue sample. However, under some circumstances they take multiple tissue samples from the same individual. But they don't have a column in the data set with an indvividual ID. So, when the send their samples to people who will genotype them, a lot of individuals are unwittingly genotyped twice. Their response: "Well, isn't it obvious that if a tissue sample is taken from a fish that was caught on the same `vessel` on the same `day` and in the same `haul`, and is of the same `species` and has the same recorded `length` is the same individual?" My response: "**NO! It isn't!**. Don't make people use a whole lot of columns to identify things that should be identified in a single column!"

6.4 An example of using some joins

Let's walk through a simple case that should be familiar to those in the group who have worked with genetic data and have had to deal with the problem of attaching meta data to genetic data coming off a genotyping instrument.

Typically those data come out in a form that can be made into a tibble. Let's read in a toy example:

```
library(tidyverse)
genos <- read_csv("inputs/toy_geno.csv")
genos
```

```
## # A tibble: 8 × 5
##     bird locus1_a locus1_b locus2_a locus2_b
##    <chr>    <int>    <int>    <int>    <int>
## 1 wiwa01        1        2        3        3
## 2 wiwa02        2        2        4        4
## 3 wiwa03        2        2        3        4
## 4 wiwa04        1        1        4        4
## 5 wiwa05        2        2        3        4
## 6 wiwa06        1        1        4        4
## 7 wiwa07        1        2        3        4
## 8 wiwa08        1        1        3        4
```

There is a single column (`bird` in this case) that is the primary key that uniquely identifies individuals. Then each locus gets two columns of data (one for each gene copy in a diploid).

That is all well and good. But now, consider this problem: for a particular analysis we are going to do, we need to have the latitude and longitude coordinates where each bird was sampled. Let's say that we got these samples from friendly collectors who provided a meta data file that gave us the collection location and the name of the collector. Let's look at that:

```
meta <- read_csv("inputs/toy_meta.csv")
meta
```

```
## # A tibble: 10 × 3
##    field_id location collector
##       <chr>    <chr>     <chr>
## 1    wiwa01     swAZ       joe
## 2    wiwa02     nwAZ      mary
## 3    wiwa03     soCA       ted
## 4    wiwa05     soCA       ted
## 5    wiwa06     swAZ       joe
## 6    wiwa08     noCA      erin
## 7    wiwa09     noCA      erin
## 8    wiwa10     noCA      erin
## 9    wiwa11     noCA      erin
## 10   wiwa12     noCA      erin
```

Notice: there are some birds in this data set that we don't have in the `genos` tibble. Not only that, but if you look closely, it is missing some birds in `genos`: they are "wiwa04" and "wiwa07". Also, note that the column that holds the ID of each bird is called `field_id` not `bird`.

Finally, the network of bird sample collectors maintains a data base of all their location codes that looks like this:

```
locations <- read_csv("inputs/toy_locations.csv")
locations
```

```
## # A tibble: 6 × 3
##   location   lat   long
##      <chr> <dbl>  <dbl>
## 1     swAZ  32.1 -112.8
## 2     nwAZ  36.8 -113.5
## 3     soCA  33.2 -117.1
## 4     noCA  40.6 -123.9
## 5     soOR  42.9 -123.7
## 6     noOR  45.9 -123.1
```

Aha! So, what we need to do is associate with each `bird` a `location`, and then once we have done with that, we need to associate a `lat` and a `long` with those locations. This is the perfect job for a join (two of them, actually).

Note that we are focused on our birds, here, so we want to keep them all around and not add any information where we don't have a bird. Hence `left_join()` is our go-to friend there (as it almost always will be).

Here is what the first step looks like:

```
genos %>%
  left_join(., meta, by = c("bird" = "field_id"))
```

```
## # A tibble: 8 × 7
##     bird locus1_a locus1_b locus2_a locus2_b location collector
##    <chr>    <int>    <int>    <int>    <int>    <chr>     <chr>
```

```
## 1 wiwa01        1        2        3        3     swAZ       joe
## 2 wiwa02        2        2        4        4     nwAZ      mary
## 3 wiwa03        2        2        3        4     soCA       ted
## 4 wiwa04        1        1        4        4     <NA>      <NA>
## 5 wiwa05        2        2        3        4     soCA       ted
## 6 wiwa06        1        1        4        4     swAZ       joe
## 7 wiwa07        1        2        3        4     <NA>      <NA>
## 8 wiwa08        1        1        3        4     noCA      erin
```

Notice that we have some NAs for birds that are not in the meta data. That is the behavior we expect from `left_join()`: it is not going to discard some of your birds, just because they don't appear in the meta data.

Note also that when we explicitly give the names of the keys (which differ in the different tibbles) the one on the left corresponds to the `x` argument to `left_join()`. Also, notice that these key names *must be quoted*!! (It is easy to forget that, because you so seldom need quotation marks around things in the tidyverse.)

Now, we can add the lat-longs on there. We will show how that is done by chaining onto the previous command:

```
genos %>%
  left_join(., meta, by = c("bird" = "field_id")) %>%
  left_join(., locations)
```

```
## Joining, by = "location"
```

```
## # A tibble: 8 × 9
##     bird locus1_a locus1_b locus2_a locus2_b location collector   lat
##    <chr>    <int>    <int>    <int>    <int>    <chr>     <chr> <dbl>
## 1 wiwa01        1        2        3        3     swAZ       joe  32.1
## 2 wiwa02        2        2        4        4     nwAZ      mary  36.8
## 3 wiwa03        2        2        3        4     soCA       ted  33.2
## 4 wiwa04        1        1        4        4     <NA>      <NA>    NA
## 5 wiwa05        2        2        3        4     soCA       ted  33.2
## 6 wiwa06        1        1        4        4     swAZ       joe  32.1
## 7 wiwa07        1        2        3        4     <NA>      <NA>    NA
## 8 wiwa08        1        1        3        4     noCA      erin  40.6
## # ... with 1 more variables: long <dbl>
```

Voila! That is what we wanted. Now, we could filter out those NAs and drop the `collector` column, if desired.

Notice that, since the `location` column was named `location` in tibble, `left_join()` just used that.

6.4.1 When would I use `right_join()`?

The only time I use this is when I want to add columns to the beginning of a tibble, but I want to preserve all the keys in the table that is going to end up with its columns on the right hand side of the table. And even then I usually just use `select` after a `left_join()`. Perhaps an example will be best: for some purposes it is best to keep the genotype data all together on the right hand side of the tibble (often genotype data can take up lots of columns and you might want to be able to see the columns you have joined on without using View() and scrolling way over). In this case you can do this:

```
genos %>%
  right_join(meta, ., by = c("field_id" = "bird")) %>%
  right_join(locations, .)
```

```
## Joining, by = "location"
```

```
## # A tibble: 8 × 9
##   location   lat   long field_id collector locus1_a locus1_b locus2_a
##      <chr> <dbl>  <dbl>    <chr>     <chr>    <int>    <int>    <int>
## 1     swAZ  32.1 -112.8   wiwa01       joe        1        2        3
## 2     nwAZ  36.8 -113.5   wiwa02      mary        2        2        4
## 3     soCA  33.2 -117.1   wiwa03       ted        2        2        3
## 4     <NA>    NA     NA   wiwa04      <NA>        1        1        4
## 5     soCA  33.2 -117.1   wiwa05       ted        2        2        3
## 6     swAZ  32.1 -112.8   wiwa06       joe        1        1        4
## 7     <NA>    NA     NA   wiwa07      <NA>        1        2        3
## 8     noCA  40.6 -123.9   wiwa08      erin        1        1        3
## # ... with 1 more variables: locus2_b <int>
```

Notice that when you do this, the column name of the `x` variable, `field_id` is the one that gets retained.

6.4.2 What about that `inner_join()`

If you knew ahead of time that you couldn't use any birds that you didn't have lat-longs for, you could start with an `inner_join()`, because that would discard birds that don't have an entry in the meta data:

```
genos %>%
  inner_join(., meta, by = c("bird" = "field_id"))
```

```
## # A tibble: 6 × 7
##     bird locus1_a locus1_b locus2_a locus2_b location collector
##    <chr>    <int>    <int>    <int>    <int>    <chr>     <chr>
## 1 wiwa01        1        2        3        3     swAZ       joe
## 2 wiwa02        2        2        4        4     nwAZ      mary
## 3 wiwa03        2        2        3        4     soCA       ted
## 4 wiwa05        2        2        3        4     soCA       ted
## 5 wiwa06        1        1        4        4     swAZ       joe
## 6 wiwa08        1        1        3        4     noCA      erin
```

But, it will probably be easier to follow if you *explicitly discard those birds* using `filter()` or by doing a filtering join.

6.4.3 An anti_join example

If you want to get all the genotype data for birds that don't occur in the meta data you can use the filtering `anti_join()`:

```
genos %>%
  anti_join(., meta, by = c("bird" = "field_id"))
```

```
## # A tibble: 2 × 5
##     bird locus1_a locus1_b locus2_a locus2_b
##    <chr>    <int>    <int>    <int>    <int>
## 1 wiwa07        1        2        3        4
## 2 wiwa04        1        1        4        4
```

And to return only those rows for birds that are in the meta data, you could use `semi_join()`:

```
genos %>%
  semi_join(., meta, by = c("bird" = "field_id"))
```

```
## # A tibble: 6 × 5
##     bird locus1_a locus1_b locus2_a locus2_b
```

```
##    <chr>    <int>    <int>    <int>    <int>
## 1 wiwa01        1        2        3        3
## 2 wiwa02        2        2        4        4
## 3 wiwa03        2        2        3        4
## 4 wiwa05        2        2        3        4
## 5 wiwa06        1        1        4        4
## 6 wiwa08        1        1        3        4
```

Note that we could turn it around in order to see which birds are in the meta data, but which don't occur in the genos:

```
genos %>%
  anti_join(meta, ., by = c("field_id" = "bird"))
```

```
## # A tibble: 4 × 3
##   field_id location collector
##      <chr>    <chr>     <chr>
## 1   wiwa12     noCA      erin
## 2   wiwa11     noCA      erin
## 3   wiwa10     noCA      erin
## 4   wiwa09     noCA      erin
```

Dammit Erin! You always forget to send us the friggin' samples! Clearly smokin' too much of the kind green there in noCA. (Note. These names really are totally fictitious.)

Quick quiz: why would this not work:

```
genos %>%
   anti_join(meta, ., by = c("bird" = "field_id"))
```

6.4.4 Just for fun, let's see a full_join()

This bad boy makes a row with NAs for values in *either* tibble that are not matched in the other:

```
genos %>%
  full_join(., meta, by = c("bird" = "field_id")) %>%
  full_join(., locations) %>%
  print(n = 20) # so all rows print
```

```
## Joining, by = "location"
```

```
## # A tibble: 14 × 9
##      bird locus1_a locus1_b locus2_a locus2_b location collector   lat
##     <chr>    <int>    <int>    <int>    <int>    <chr>     <chr> <dbl>
## 1  wiwa01        1        2        3        3     swAZ       joe  32.1
## 2  wiwa02        2        2        4        4     nwAZ      mary  36.8
## 3  wiwa03        2        2        3        4     soCA       ted  33.2
## 4  wiwa04        1        1        4        4     <NA>      <NA>    NA
## 5  wiwa05        2        2        3        4     soCA       ted  33.2
## 6  wiwa06        1        1        4        4     swAZ       joe  32.1
## 7  wiwa07        1        2        3        4     <NA>      <NA>    NA
## 8  wiwa08        1        1        3        4     noCA      erin  40.6
## 9  wiwa09       NA       NA       NA       NA     noCA      erin  40.6
## 10 wiwa10       NA       NA       NA       NA     noCA      erin  40.6
## 11 wiwa11       NA       NA       NA       NA     noCA      erin  40.6
## 12 wiwa12       NA       NA       NA       NA     noCA      erin  40.6
## 13   <NA>       NA       NA       NA       NA     soOR      <NA>  42.9
```

```
## 14   <NA>       NA       NA       NA       NA     noOR      <NA>  45.9
## # ... with 1 more variables: long <dbl>
```

This is not typically what we want! Sometimes it is…but usually you will be using `left_join()`.

6.5 Working with R Notebooks

R Notebooks are totally awesome! They combine the nice features of working at the R console (namely having access to variables that remain in your .GlobalEnvironment), with the beauty of being able to document things in an easy to read and digest RMarkdown format.

Here, you can download an example of a short Notebook from one of the projects that Kristen and Eric are working on: choosing-snps

R notebooks are RMarkdown documents using the `html_document` option. They are great for doing and explaining analyses. This is where I end up doing most of my analyses these days. Typically I put them in a directory called `R-main` in my project. This is a more appropriate place to put long analyses than in README.Rmd. The README.Rmd file should be reserved for describing your data and giving people instructions on how to conduct the analysis, e.g. "Open ./R-main/01-clean-data.Rmd and run it all. Then run all the code in 02-compute-statistics.Rmd, etc."

6.5.1 Open your own R Notebook

This is easy in RStudio: File -> New File -> R Notebook. This gives you a simple template that lets you see how things work and into which you can insert your own thoughts and writings.

For quick help with formatting: Help -> Markdown Quick Reference

For more Markdown info: Help -> Cheatsheets -> R Markdown Cheat Sheet

6.5.2 Working with R Notebooks

Big difference from "regular" R Markdown documents: there is no "Knit" button.

Instead, to get results from the code, you must evaluate it, then "Preview".

Ways of evaluating code blocks:

1. right-facing triangle – evaluate current block
2. down-facing gray triangle – evaluate all the blocks *above* this one.

Or you can use keyboard shortcuts, or use the "Run" button in the upper right of the document.

Results from code blocks get presented in the Notebook.

To get something like "Knit": do this:

Run -> Restart R and Run All Chunks

6.5.3 Some caveats about notebooks

1. You should restart and Run All occasionally to make sure it is reproducible.
2. When evaluating code within an R notebook, by default the working directory for R is set to the directory that the R notebook live in, not the root directory of the project. So, it can give different results (for example when doing file acess) than the R Console. This can be hugely frustrating.

3. your variables all live in the GlobalEnvironment, so they are at risk of getting overwritten if you use the same variable name in another Notebook that you are working on at the moment. For this reason, to check reproducibility, occasionaly check that Run -> Restart R and Run All Chunks works for you.

The .nb.html files don't play very well with GitHub. If you want to share them, they are great for emailing to people (but tell them to download it and view it as a file—the gmail viewer does a crappy job of rendering it.)

6.5.4 Opening a .nb.html file in Rstudio

Doing this "reconstitutes" the .Rmd file that made it (along with the results that are saved in it). Which is sort of cool. However, it does not reconstitute all the data, etc. that went into it. So, y'all wouldn't be able to run 02-choosing-96-SNPs.Rmd.

6.6 For Next Week:

Read Chapter 3: Data Visualization

Chapter 7

Week 6 meeting: using ggplot2

Today we are going to do a super brief review of the book chapter, namely just this template:

```
ggplot(data = <DATA>) +          # DATA is a tibble of data *in long (tidy) format*
  <GEOM_FUNCTION>(               # geom_point(), geom_bar(), geom_histogram(), etc.
                                 # each geom_function output gets layered on top of the last!
     mapping = aes(<MAPPINGS>), # mappings wrapped up in the aes() function
     stat = <STAT>,              # usually can be avoided if the default is used
     position = <POSITION>       # dodge, stack, jitter...
  ) +
  <COORDINATE_FUNCTION> +        # coord_flip(), or coord_quickmap(), or other things to set scales.
  <FACET_FUNCTION>               # facet_grid() or facet_wrap()
```

And now we are going to launch directly into having everyone try this out on their own.

Here is a data set of simulation results:

```
library(tidyverse)
sims <- readRDS("inputs/alewife_snps_100_of_200.rds")
```

That looks like this:

```
sims
```

```
## # A tibble: 700 × 6
##     iter     repunit   true_rho   rho_mcmc     rho_bh     rho_pb
##    <int>      <fctr>      <dbl>      <dbl>      <dbl>      <dbl>
## 1      1         CAN 0.04573827 0.06489907 0.04550360 0.06226238
## 2      1         NNE 0.06962820 0.07385272 0.05886538 0.06360804
## 3      1          MB 0.14588315 0.21344097 0.21242781 0.22291767
## 4      1         NUN 0.46891013 0.14646259 0.35338886 0.24710874
## 5      1         BIS 0.06644381 0.09336929 0.15767998 0.08005169
## 6      1         LIS 0.15719031 0.35012560 0.11664560 0.27002139
## 7      1 MidAtlantic 0.04620614 0.05784975 0.05548877 0.05403009
## 8      2         CAN 0.02751659 0.02866411 0.02224521 0.02645824
## 9      2         NNE 0.23190134 0.27633524 0.20446014 0.26219858
## 10     2          MB 0.16320168 0.07701908 0.17914736 0.11657221
## # ... with 690 more rows
```

What we have here is a true proportion (`true_rho`) that was simulated (100 iterations for each of 7 `repunits`), and then we have estimates of that proportion using different methods (`rho_mcmc`, `rho_bh`, and `rho_pb`).

We want to compare the different types of estimated rho's to the `true_rho` for each repunit.

But before we do this, there is one more thing we need to know about ggplot: like most of the tidyverse tools, it doesn't like having multiple columns *of the same variable*. What do we mean by that? Well, `rho_mcmc`, `rho_bh`, and `rho_pb` are all just different flavors of a variable we might call `estimated_rho`.

We won't get to it this quarter, but everyone should read the Tidy Data chapter. It will show how and why we would want to tidy our data to look like this:

```
tidy_sims <- sims %>%
  tidyr::gather(data = ., key = "estimation_method", value = "estimated_rho", rho_mcmc:rho_pb)

# see what that looks like:
tidy_sims %>%
  arrange(iter, repunit, estimation_method)
```

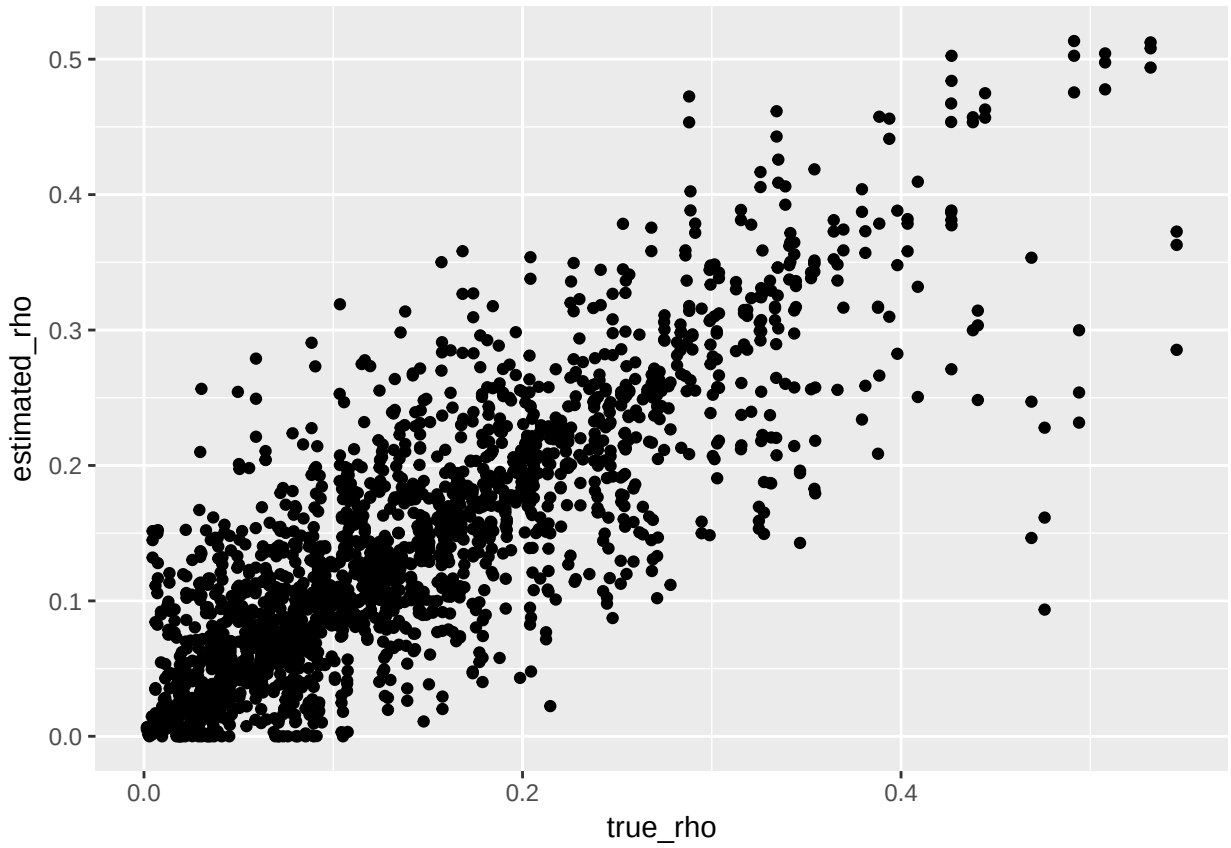
```
## # A tibble: 2,100 × 5
##     iter repunit   true_rho estimation_method estimated_rho
##    <int>  <fctr>      <dbl>             <chr>         <dbl>
## 1      1     CAN 0.04573827            rho_bh    0.04550360
## 2      1     CAN 0.04573827          rho_mcmc    0.06489907
## 3      1     CAN 0.04573827            rho_pb    0.06226238
## 4      1     NNE 0.06962820            rho_bh    0.05886538
## 5      1     NNE 0.06962820          rho_mcmc    0.07385272
## 6      1     NNE 0.06962820            rho_pb    0.06360804
## 7      1      MB 0.14588315            rho_bh    0.21242781
## 8      1      MB 0.14588315          rho_mcmc    0.21344097
## 9      1      MB 0.14588315            rho_pb    0.22291767
## 10     1     NUN 0.46891013            rho_bh    0.35338886
## # ... with 2,090 more rows
```

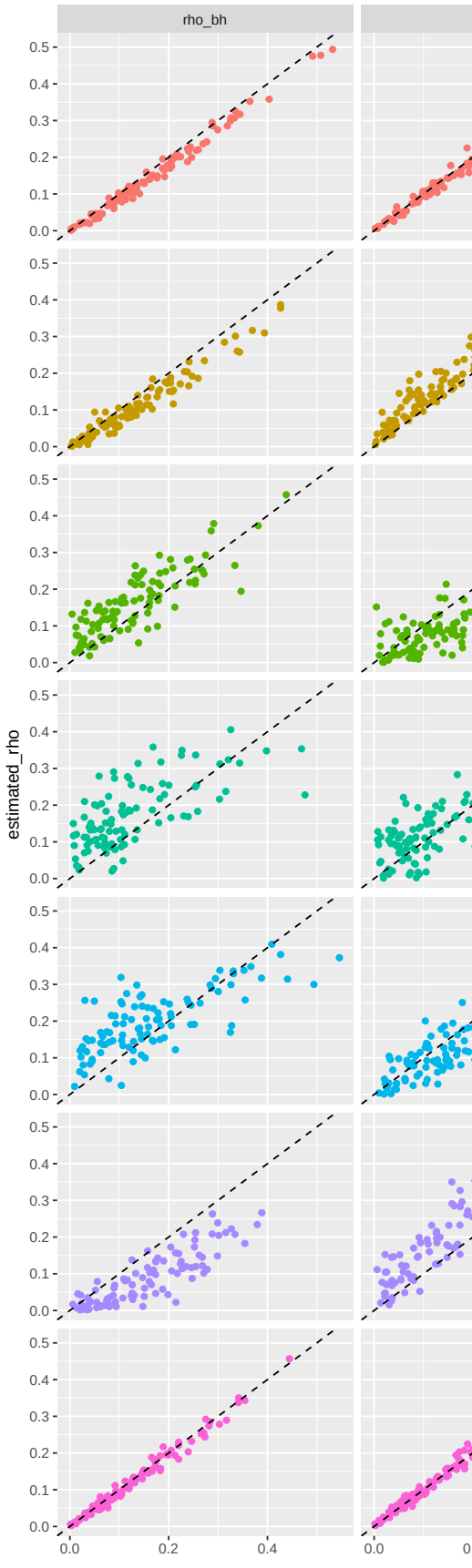
See that we have taken the three columns of estimates and put them int he `estimated_rho` column, and we are using the `estimation_method` column to say which method was used for each row of values in `estimated_rho`.

You can download that `tidy_sims` tibble from here.

Now, we could start by making a scatter plot of estimated versus true rho with ggplot. That would look like this:

```
ggplot(tidy_sims, aes(x = true_rho, y = estimated_rho)) +
  geom_point()
```


estimated_rho
0.0
0.1
0.2
0.3
0.4
0.5
true_rho
0.0
0.2
0.4



OK, that is relatively uninformative. How about we spice it up a bit to make this:

7.0.1 Your Mission

OK Class! Here is your mission.

1. Download `tidy_sims` and figure out during our class time how to make the above plot.
   Hints: you will be using the `colour` aesthetic, a `geom_abline()` geom, and also `facet_grid()`
2. When you have finished that, use your newly-acquired ggplot skills to make an informative plot from your own data set.

I will be here in class to help with this and field any questions.

7.1 Some lecture notes from a few years ago

This is just a rewrite of some playing with ggplot that I did for the reproducible research course a few years ago, updated to the tidyverse.

It might be worth a reading through to see more examples ### Prerequisites {#ggplot-prereq} * To work through the examples you will need another package that you might not have yet. * Please download/install these before coming to class: 1. install necessary packages:

    ```r
    install.packages("lubridate")
    ```

2. Pull the most recent version of the rep-res-course repo just before coming to class.

7.1.1 Goals for this hour:

1. Describe (briefly) `ggplot2`s underlying philosophy and how to work with it.
2. Quickly overview the _geom_s available in `ggplot2`
3. Develop an example plot together
4. Turn you all loose with the `mosaic` package to experiment with different plots

7.2 About ggplot2

7.2.1 Basics

- A package created by Hadley Wickham in 2005
- Implements Wilkinson's "grammar of graphics"
- Unified way of thinking about 2-D statistical graphics
- Not entirely easy to learn
  - if you already know R's base graphics system, it is painful to re-learn a different way of doing things
  - if you don't already know how to do graphics in R, be glad.
  - regardless it is worth learning ggplot
  - I am not even going to teach R's base graphics system
- Amazing for quick data exploration and also produces publication quality graphics
- Support for legends etc., considerably better/easier than R base graphics

7.2.2 What is this grammar of graphics?

- Traditionally, people have referred to plots by *name*
  - i.e., scatterplot, histogram, bar chart, bubble plot, etc.

- Disadvantages:
  - Lots of possible graphics = way too many names
  - Fails to acknowledge the common elements / similarities / dissimilarities between different plots
- Wilkinson's *Grammar of Graphics* (a book) describes a few building blocks which when assembled together in particular ways can generate all these named graphics (and more)
  - Provides a nice way of thinking about and describing graphics

7.2.3 ggplot2

- Hadley Wickham's R implementation of a modified (*layered*) grammar of graphics
- `ggplot` and `ggplot2` are similar. 'ggplot2 is just more recent (and recommended)
- `ggplot` operates on *data frames*
  - in R base graphics typically you pass in vectors
  - in `ggplot` everything you want to use in a graphic must be contained within a data frame
  - Takes getting used to, but ultimately is a good way of thinking about it.

7.2.4 Components of the grammar of graphics

1. *data* and *aesthetic mappings*
2. *geoms* (geometric objects)
3. *stats* (statistical transformations)
4. *scales*
5. *coords* (coordinate systems)
6. *facets* (a specification of how to break things into smaller subplots)

We will focus on 1, 2 for most of today.

7.2.5 In a nutshell

Without getting into the complications of scales and coordinate systems here, in a nutshell, is what ggplot does:

- Layers in plots are made by:
  1. mapping *values* in the columns of a data frame to *aesthetics*, which are properties that can visually express differences, for example:
     a. x-position
     b. y-position
     c. shape (of a plot character, for example)
     d. color
     e. size (of a point, for example)
  2. Portraying those values by drawing a *geometric object* whose appearance and placement in space is dictated by the mapping of values to aesthetics.

7.3 An example, please

Phew! That is a crazy mouthful. Is this really going to help us make pretty plots?

All I can say is you owe it to yourself to persevere — `ggplot2` is really worth the effort!

7.3.1 A pole vaulting example

- Here is a concrete example: we will investigate the history of pole-vaulting world records
- I grabbed the data by copying them from http://en.wikipedia.org/wiki/Men's_pole_vault_world_record_progression and pasting them into a text file
- Here we make a data frame out of them:

```
library(tidyverse)  # for dealing with dates
library(stringr)
library(lubridate)  # functions mdy, ymd, today(), etc.



# first off read the data into a data frame
pv <- read_tsv("inputs/mens_pole_vault_raw.txt", comment = "%") %>%
  mutate(Date = lubridate::mdy(str_replace(Date, "\\[[0-9]\\]", ""))) %>% # clean up dates
  mutate(Meters = parse_number(Record))  # get the record in a numeric number of meters
```

```
## Parsed with column specification:
## cols(
##   Record = col_character(),
##   Athlete = col_character(),
##   Nation = col_character(),
##   Venue = col_character(),
##   Date = col_character(),
##   `#[2]` = col_integer()
## )
```

- Great, what do these look like? Try `View(pv)` if you are following along. Here are the first few rows too:

```
pv
```

```
## # A tibble: 72 × 7
##    Record        Athlete        Nation               Venue       Date
##     <chr>          <chr>         <chr>               <chr>     <date>
## 1  4.02 m    Marc Wright United States     Cambridge, U.S. 1912-06-08
## 2  4.09 m     Frank Foss United States    Antwerp, Belgium 1920-08-20
## 3  4.12 m   Charles Hoff        Norway Copenhagen, Denmark 1922-09-22
## 4  4.21 m   Charles Hoff        Norway Copenhagen, Denmark 1923-07-22
## 5  4.23 m   Charles Hoff        Norway        Oslo, Norway 1925-08-13
## 6  4.25 m   Charles Hoff        Norway      Turku, Finland 1925-09-27
## 7  4.27 m     Sabin Carr United States  Philadelphia, U.S. 1927-05-27
## 8  4.30 m     Lee Barnes United States        Fresno, U.S. 1928-04-28
## 9  4.37 m William Graber United States     Palo Alto, U.S. 1932-07-16
## 10 4.39 m    Keith Brown United States        Boston, U.S. 1935-06-01
## # ... with 62 more rows, and 2 more variables: `#[2]` <int>, Meters <dbl>
```

7.3.2 A first ggplot

- There is a simplified ggplot function called `qplot` that behaves more like R's base graphics function `plot()`.
  - I don't recommend `qplot`. It will just lengthen the time it takes to understand the grammar of graphics.
  - Instead, we will use the full `ggplot()` standard syntax.

1. First we have to essentially establish a plotting area upon which to add layers. We will do this like so:

```
g <- ggplot()
```

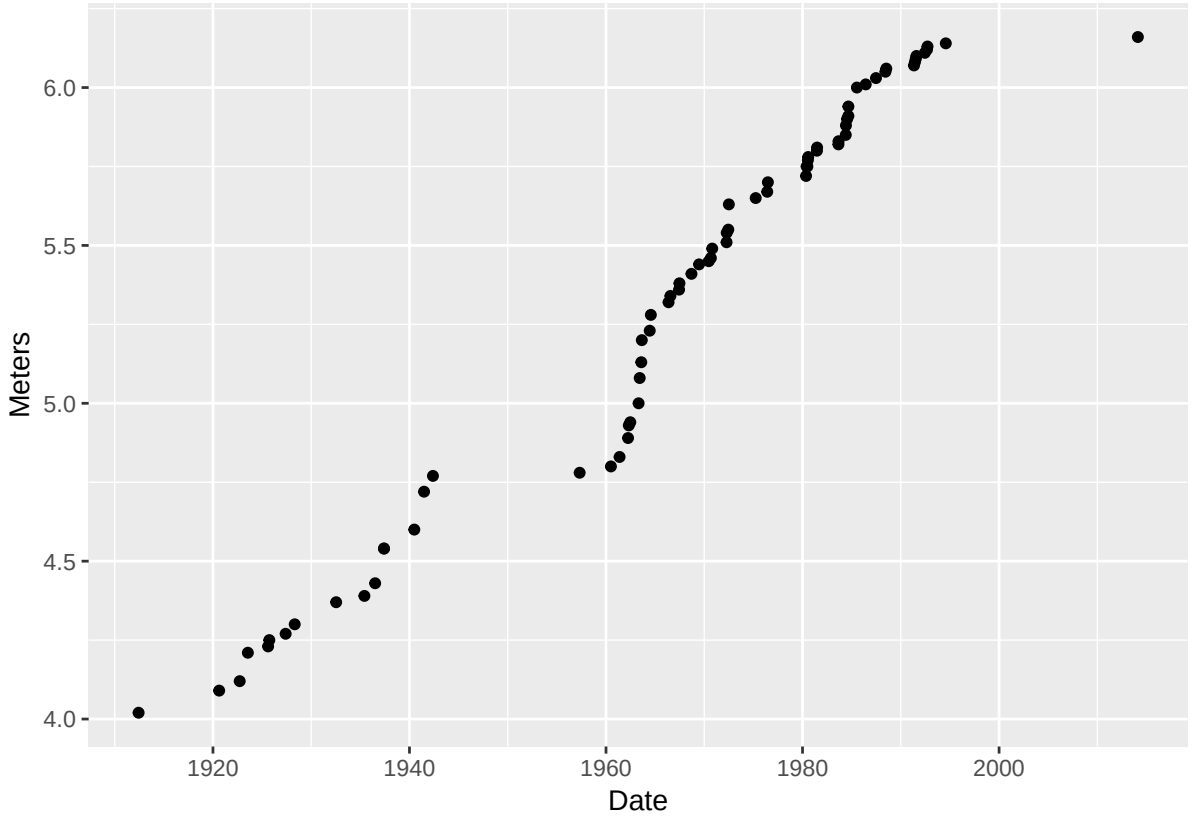
At this point, g is a ggplot plot object. We can try printing it:

```
g
```


That doesn't work, because there is nothing to plot. We have to add a layer to it.

2. Adding a layer is done by adding a collection of geometric objects to it using one of the `geom_xxxx` functions. Each such function requires a *data set* and a *mapping* of columns in the data set to *aesthetics*. Let's make some scatter-points: Meters as a function of Date:

```
g2 <- g + geom_point(data = pv, mapping = aes(x = Date, y = Meters))
g2
```



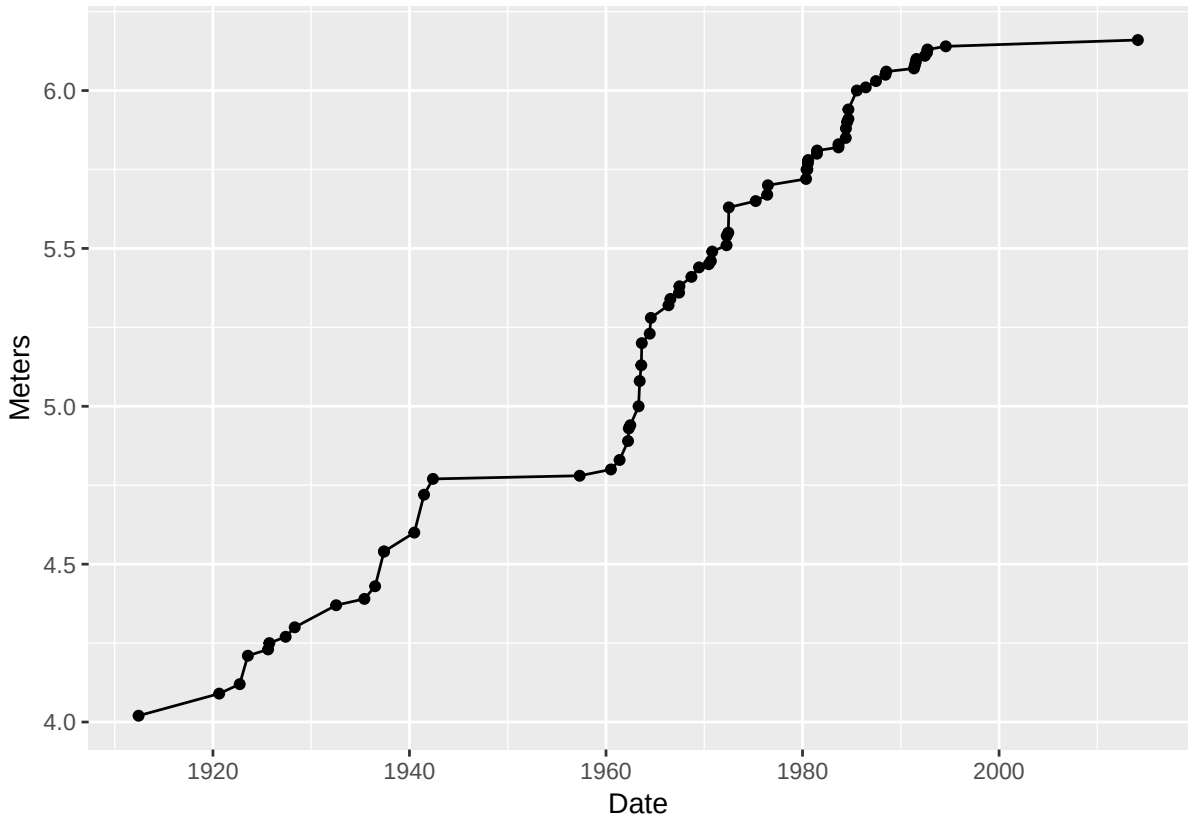
Wow! That totally worked. Here are some interesting points about:

```
g2 <- g + geom_point(data = pv, mapping = aes(x = Date, y = Meters))
g2
```

a. You add layers by catenating them with `+`.
b. the names of the columns don't need to be quoted.
c. when you map aesthestics you wrap them inside the `aes()` function
d. the full object with all the layers is returned into `g2` and then we printed it (by typing `g2`). (we could have also just said `g + geom_point(data = pv, mapping = aes(x = Date, y = Meters))`)
e. we didn't have to do anything fancy to the dates…ggplot knew how to plot them. This is thanks to turning the dates into `lubridate` objects. (If you work with dates, get to know the lubridate package!)

3. I want to overlay a line on that…No problem! Add another layer:

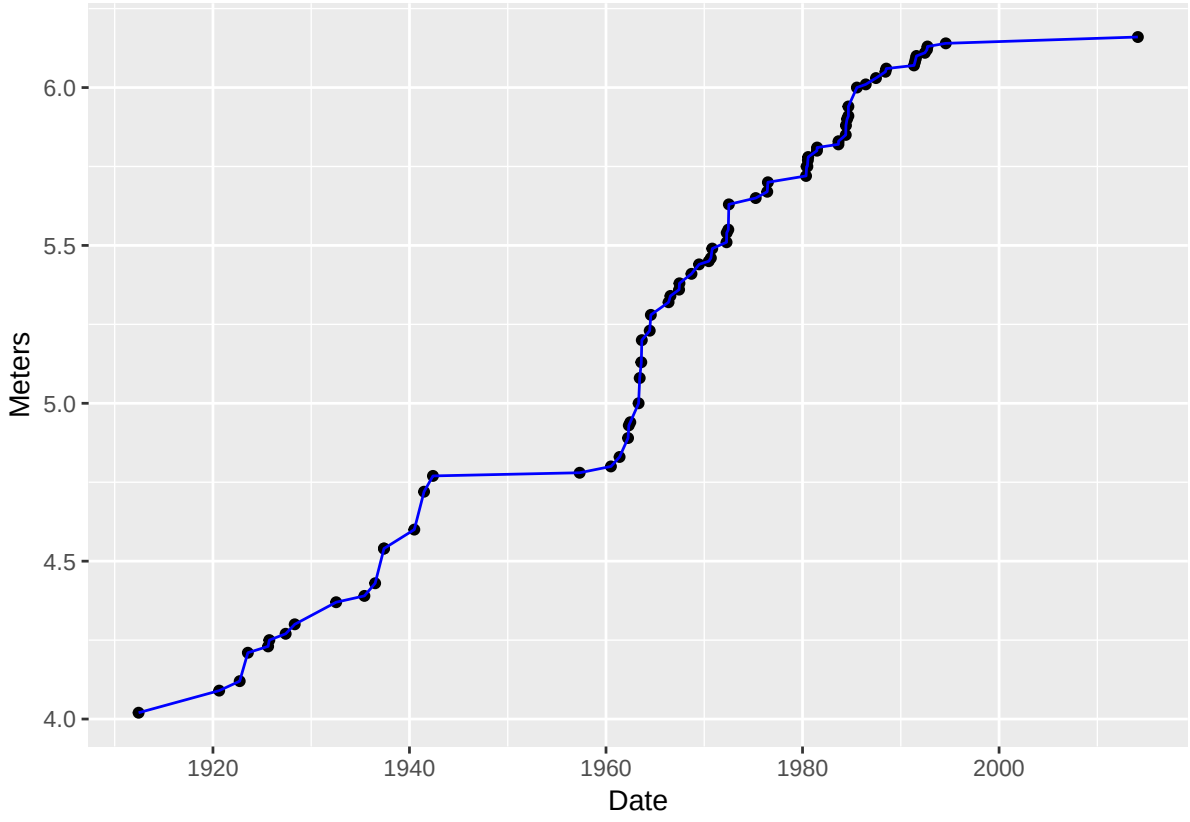
```
g3 <- g2 + geom_line(data = pv, mapping = aes(x = Date, y = Meters))
g3
```



That worked! We just added (literally, using a + sign!) another layer—one that had a line on it. BUT! what if I want to make that line blue?

4. Make the line blue. Note that you are giving the line an aesthetic property (the color blue), but you are not mapping that to any values in the data frame, **so** you don't put that within the `aes()` function:

```
g4 <- g2 + geom_line(data = pv, mapping = aes(x = Date, y = Meters), color = "blue")
g4
```



That worked! Notice that we were able to put that new layer atop `g2` which we had stored previously.

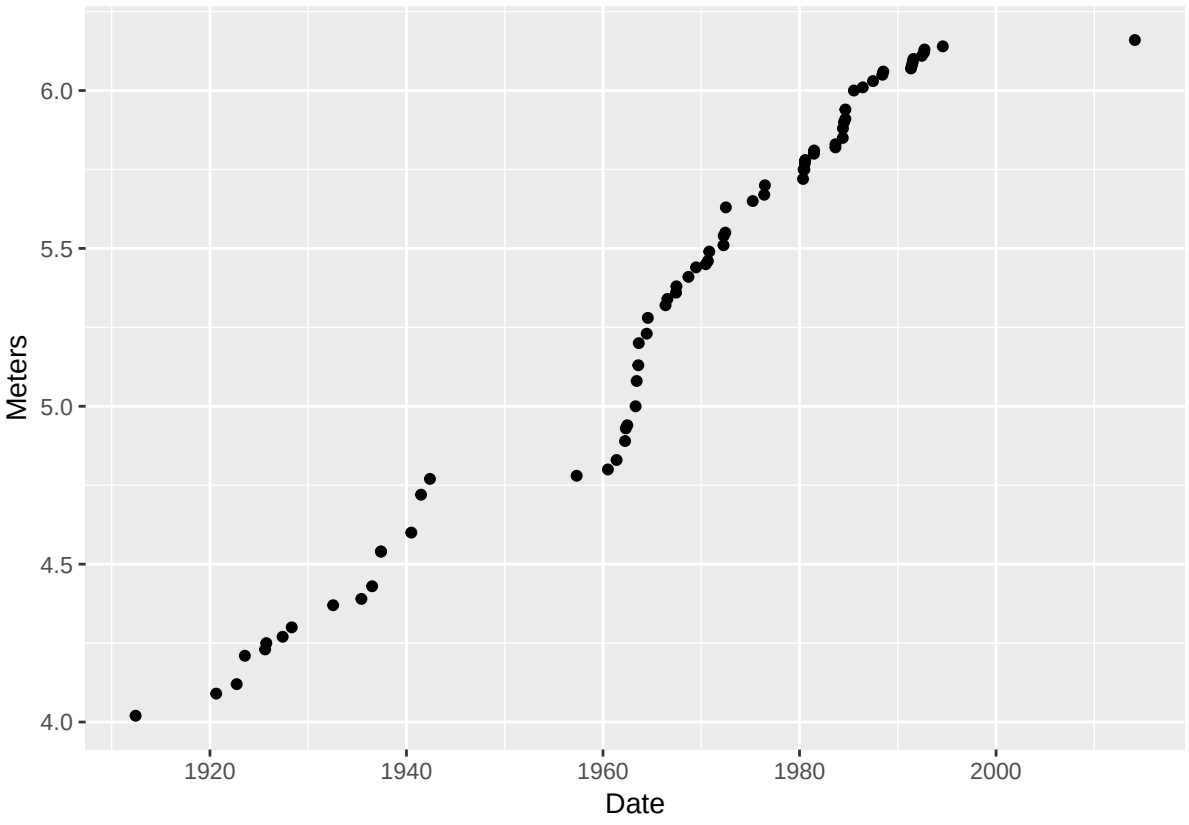
7.3.3 ggplot's system of defaults

- Hey! I am really tired of typing `data = pv, mapping = aes(x = Date, y = Meters)` isn't there some way around that?

- Yes! You can pass a default data frame and/or default mappings to the original `ggplot()` function. Then, if data and mappings are not specified in later layers, the defaults are used.

- Witness!

```
d <- ggplot(data = pv, aes(x = Date, y = Meters))  # this defines defaults
d2 <- d + geom_point()  # add a layer with points
d2  # print it
```



- Sick! Now we can add all sorts of fun layers as we see fit, each time, by invoking a `geom_xxx()` function.

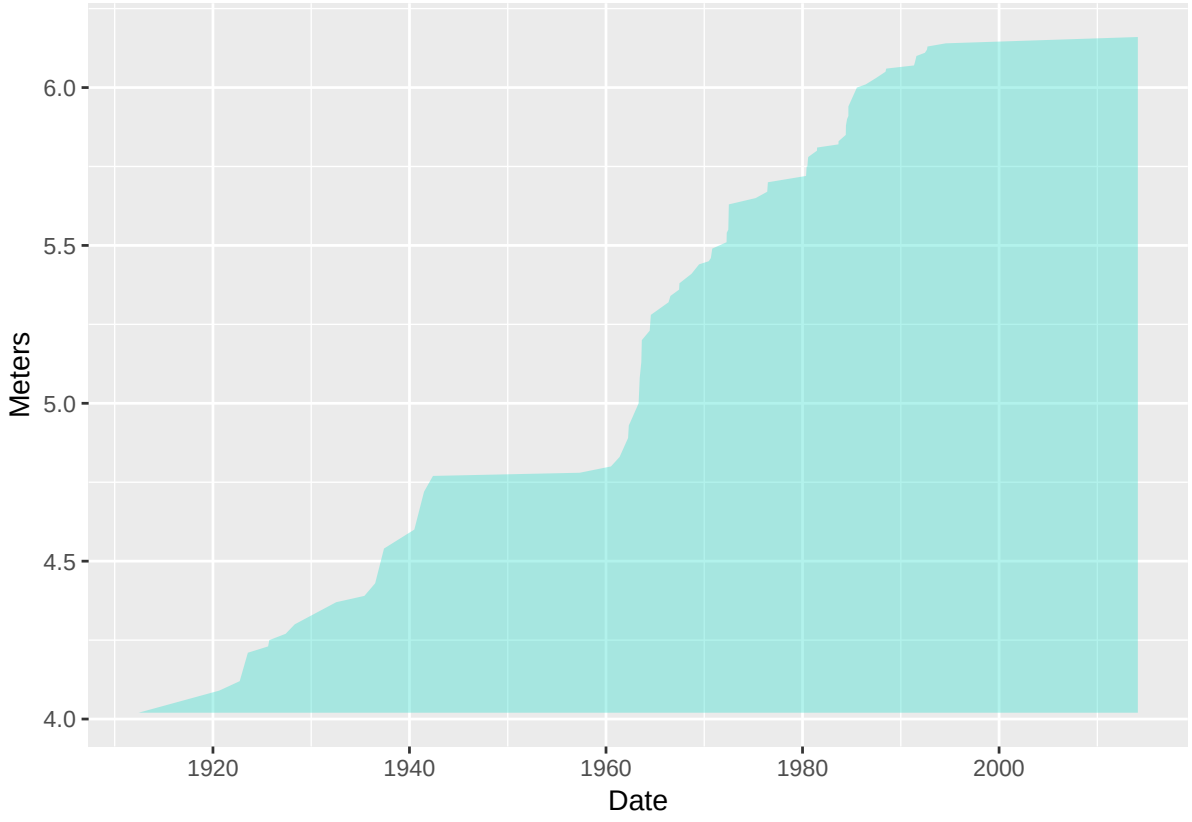
- Let's go totally crazy!

1. Establish plot base with defaults:

```
d <- ggplot(data = pv, aes(x = Date, y = Meters))
```

2. Add a transparent turquoise area along the back:

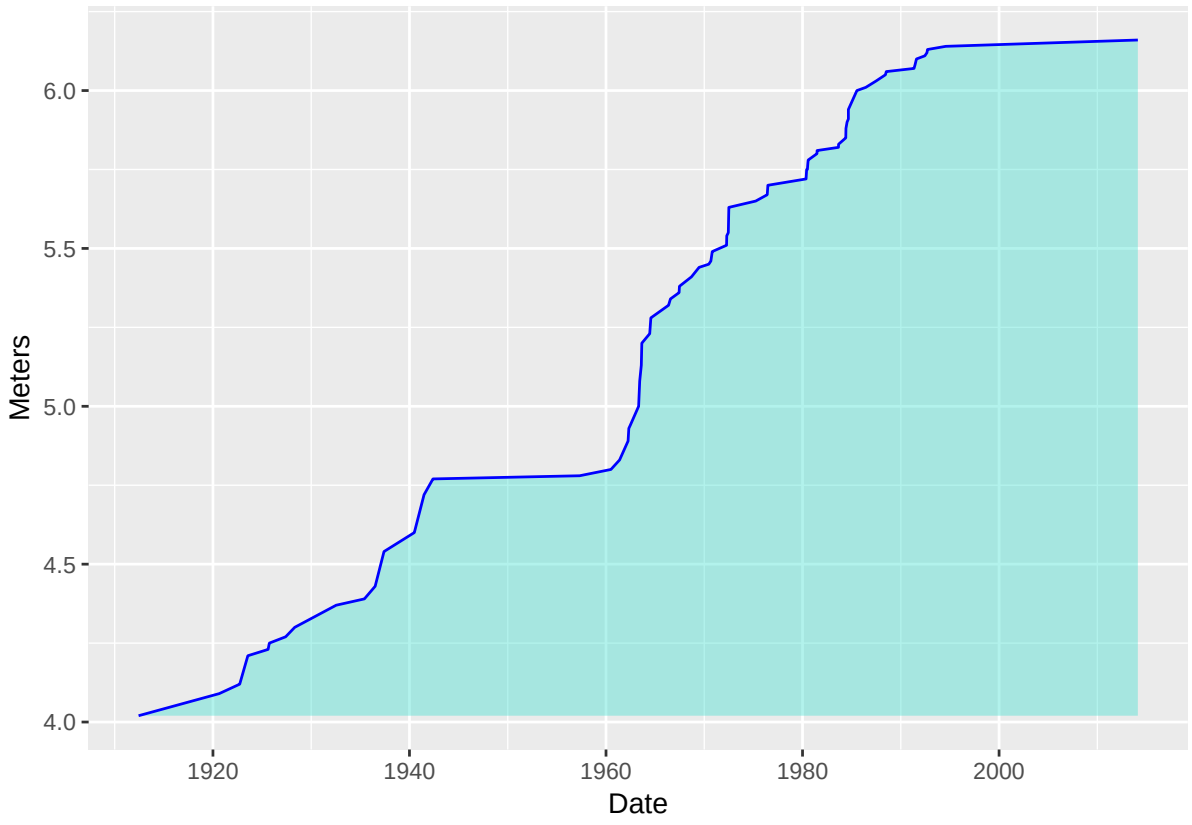
```
d2 <- d + geom_ribbon(aes(ymax = Meters), ymin = min(pv$Meters), alpha = 0.4, fill = "turquoise")
d2
```



Wow! I feel like transparency was never so easy in R base graphics!

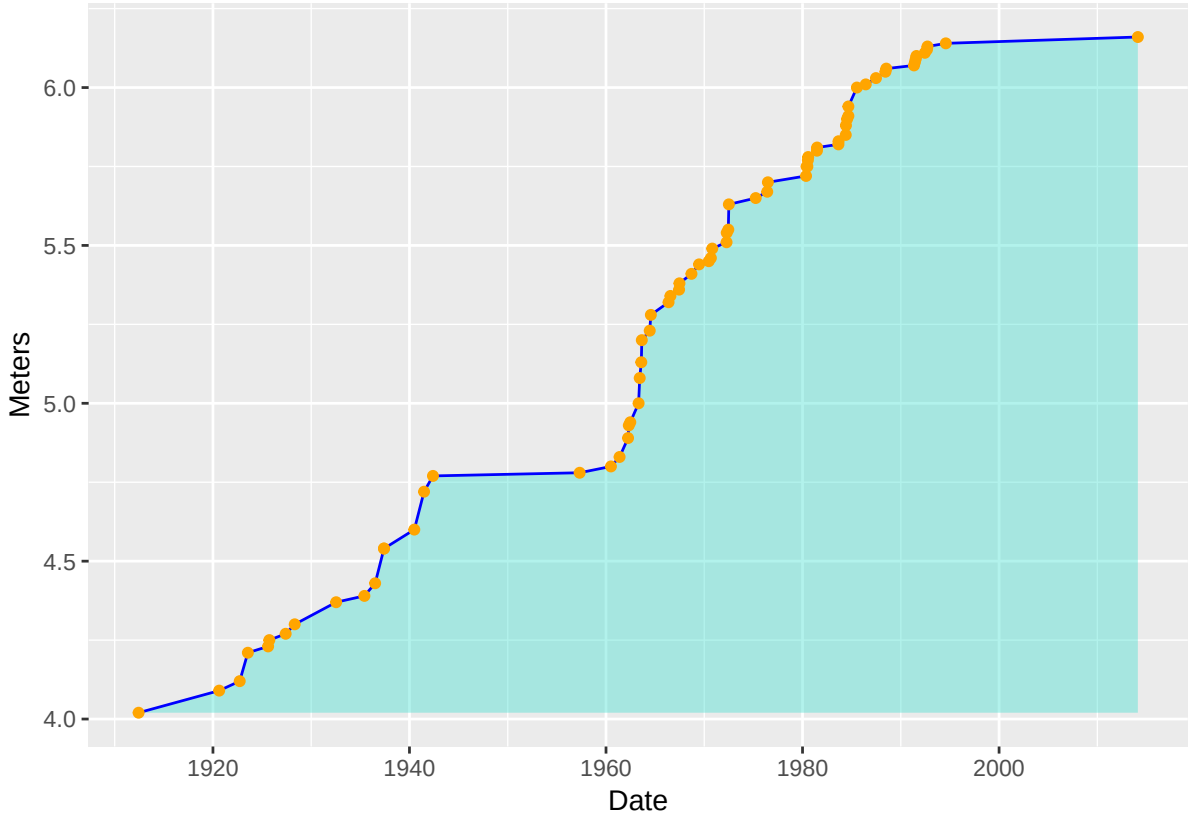
3. Put a line along there too:

```
d3 <- d2 + geom_line(color = "blue")
d3
```



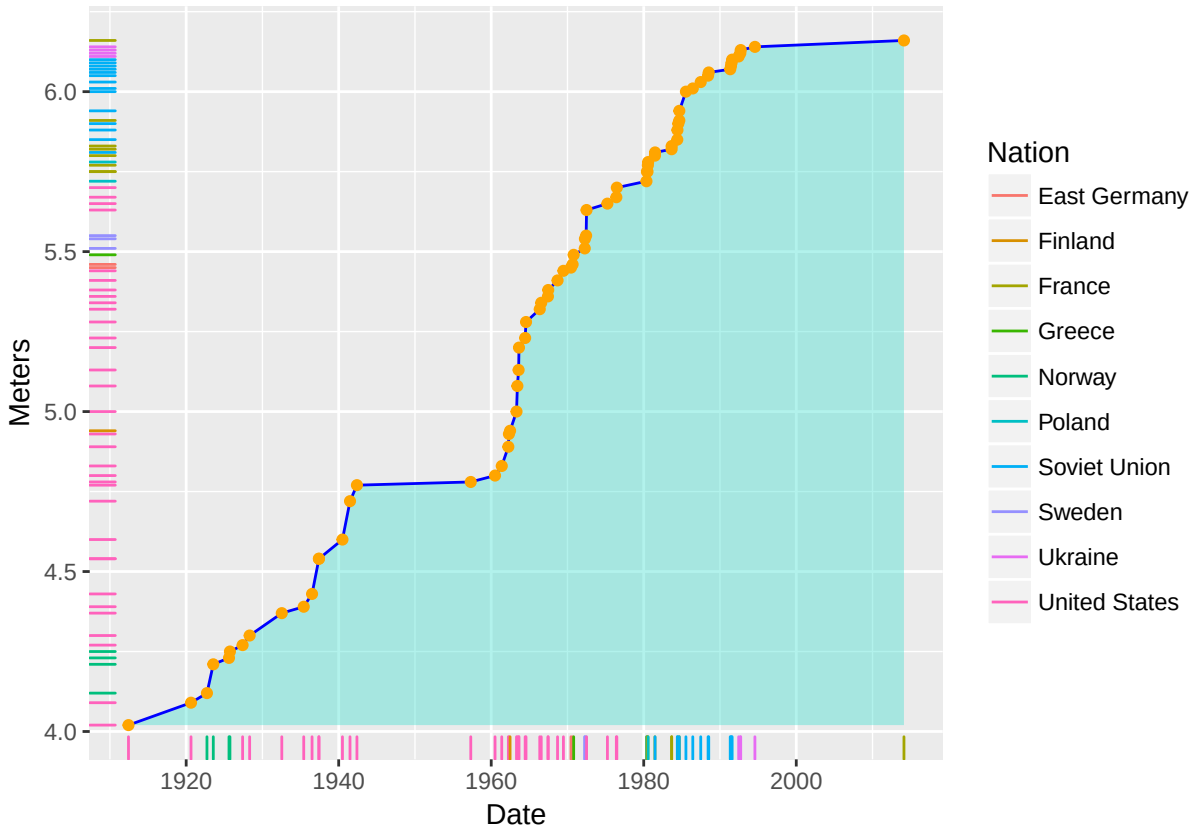
4. Now add some small orange points:

```
d4 <- d3 + geom_point(color = "orange")
d4
```



5. Now, add "rugs" along the x and y axes that show the position of points, and in them, map *color* to *Nation*:

```
d5 <- d4 + geom_rug(sides = "bl", mapping = aes(color = Nation))
d5
```



- Note that we are using the `aes()` function to add the mapping of *color* to *Nation* within the `geom_rug()` function.
  - Note also that the legend was created automatically.
- Wow! A legend is produced automatically, by default, and it looks pretty good. This doesn't happen without an all-out wrestling match in R's base graphics system.

- So, that was some fun playing around with the truly awesome ease with which you can build up complex and lovely graphics with ggplot.

7.3.4 How many geoms are there?

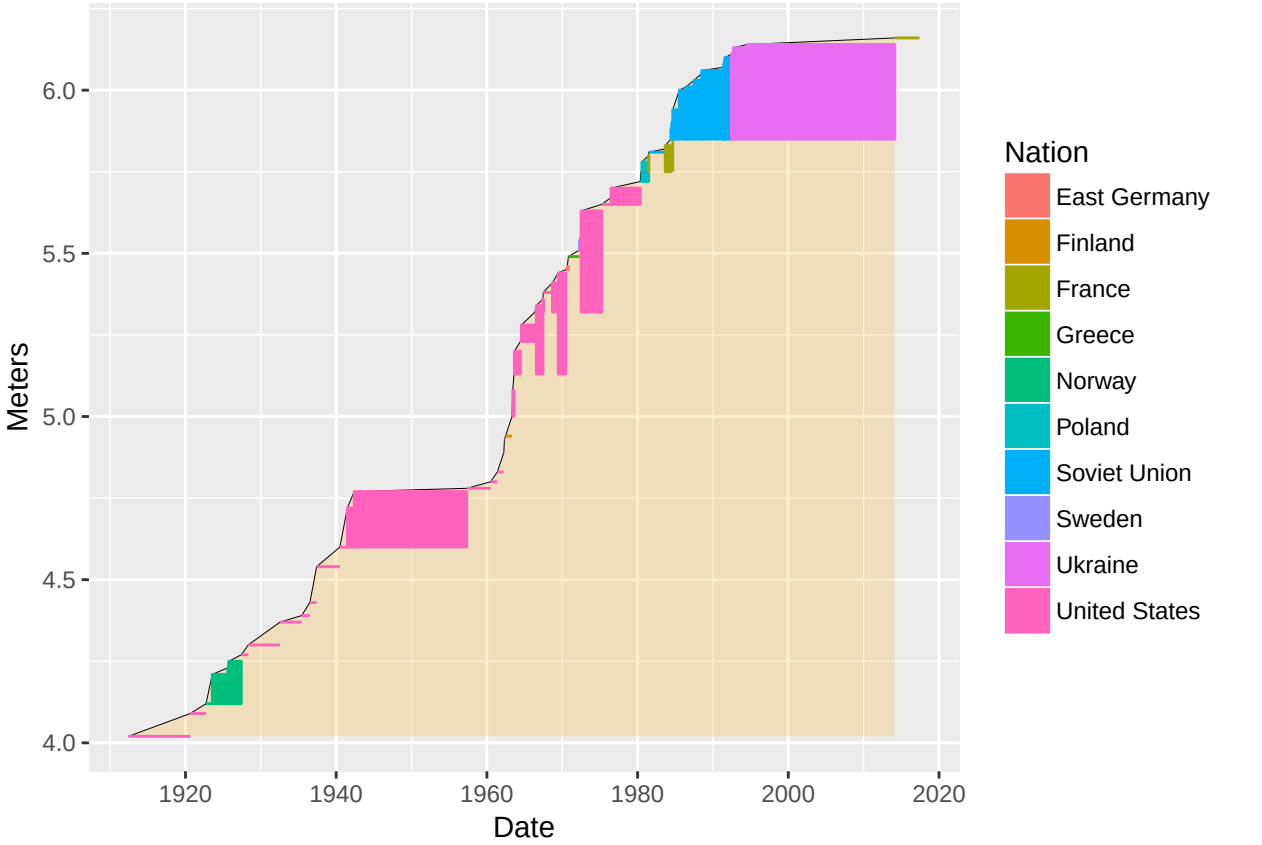
- Quite a few. There is a good summary with little icons for each here
- Note that most geoms respond to the aesthetics of `x`, `y`, and `color` (or `fill`). And some have more (or other) aesthetics you can map values to.

7.3.5 Getting even sillier

- Here is another plot I made while geeking out.
- I wanted to get a sense for how much individual athletes had improved since their first world record

```
# add a column for the date of first record for each athlete
pv <- pv %>%
  group_by(Athlete) %>%
  mutate(FirstRecordMeters = min(Meters)) %>%
  ungroup() %>%
  mutate(DateNext = lead(Date,  default = lubridate::ymd(lubridate::today())))
```

```
bb <- ggplot(data = pv, mapping = aes(x = Date, y = Meters, color = Nation))  +
  geom_ribbon(aes(ymax = Meters, color = NULL), ymin = min(pv$Meters), alpha = 0.2, fill = "orange") +
  geom_line(color = "black", size = .1) +
  geom_rect(aes(xmin = Date,
                xmax = DateNext,
                ymin = FirstRecordMeters,
                ymax = Meters,
                fill = Nation
                ))
bb
```



I used `geom_rect()` to plot rectangles where the bottom edge is situated at the athlete's lowest world record. Note how it required massaging some data into the data frame at the beginning.

7.3.6 More! Add some text to it!

Now, I want to try to get every single name in the first time the person set the record. Of course, they won't all fit exactly at the point (Date and Meters) of each record, so let's space them out evenly, then draw little line segments to meet up with the records.

1. We make a new data frame that has just the first time a person got a record

```
pv1 <- pv %>%
  filter(`#[2]` == 1)
pv1 # have a look at it
```

```
## # A tibble: 34 × 9
```

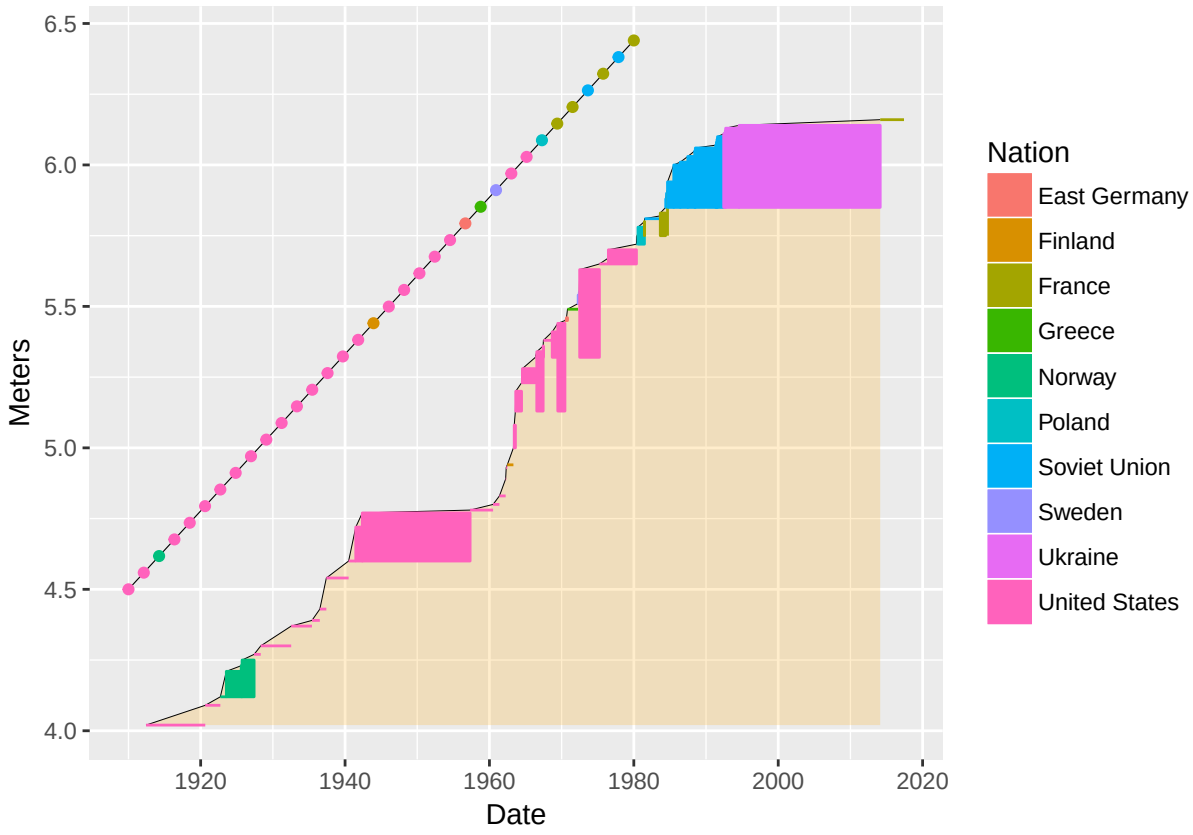
```
##    Record        Athlete        Nation                       Venue
##     <chr>          <chr>         <chr>                       <chr>
## 1  4.02 m    Marc Wright United States            Cambridge, U.S.
## 2  4.09 m     Frank Foss United States           Antwerp, Belgium
## 3  4.12 m   Charles Hoff        Norway         Copenhagen, Denmark
## 4  4.27 m     Sabin Carr United States          Philadelphia, U.S.
## 5  4.30 m     Lee Barnes United States               Fresno, U.S.
## 6  4.37 m William Graber United States            Palo Alto, U.S.
## 7  4.39 m    Keith Brown United States               Boston, U.S.
## 8  4.43 m  George Varoff United States Princeton, New Jersey, U.S.
## 9  4.54 m    Bill Sefton United States          Los Angeles, U.S.
## 10 4.54 m  Earle Meadows United States          Los Angeles, U.S.
## # ... with 24 more rows, and 5 more variables: Date <date>, `#[2]` <int>,
## #   Meters <dbl>, FirstRecordMeters <dbl>, DateNext <date>
```

2. Look at the previous plot and decide that we would like to run names equally spaced along a line that runs from about (1910, 4.25) to (1980, 6.5). Note that we will have to squish 34 names along that line. So, we can define the points along it as follows:

```
pv1 <- pv1 %>%
  mutate(nameline.x = seq(mdy("1-1-1910"), mdy("1-1-1980"), length.out = nrow(.)),
         nameline.y = seq(4.5, 6.44, length.out = nrow(.)))

pv1$nameline.x <- seq(mdy("1-1-1910"), mdy("1-1-1980"), length.out = nrow(pv1))
# holy cow! did you see how easily we got that sequence of dates? lubridate is amazing!
pv1$nameline.y <- seq(4.5, 6.44, length.out = nrow(pv1))
```

3. Add that line to the plot as colored points atop a black line.

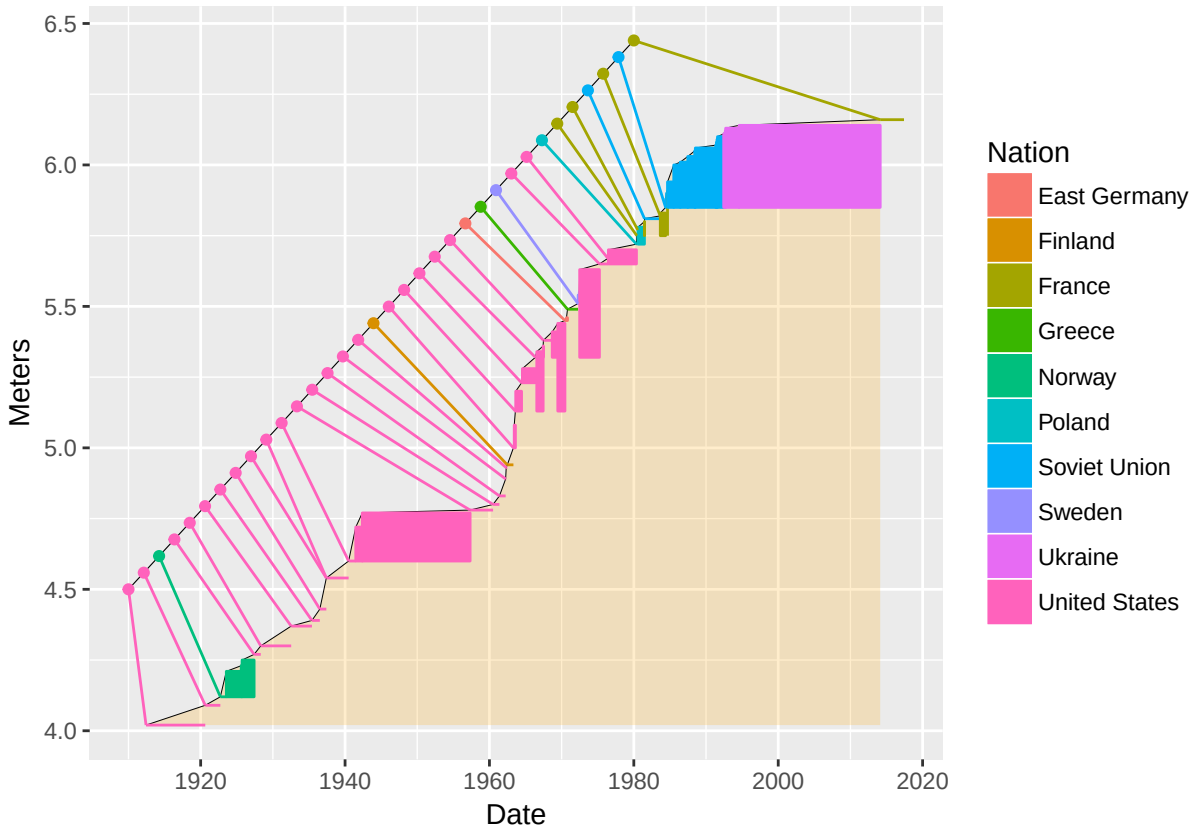
```
bb2 <- bb +
  geom_line(data = pv1, mapping = aes(x = nameline.x, y = nameline.y), color = "black", size = 0.2)
  geom_point(data = pv1, mapping = aes(x = nameline.x, y = nameline.y))
bb2
```



Notice how the color of Nation gets applied to the points automatically.

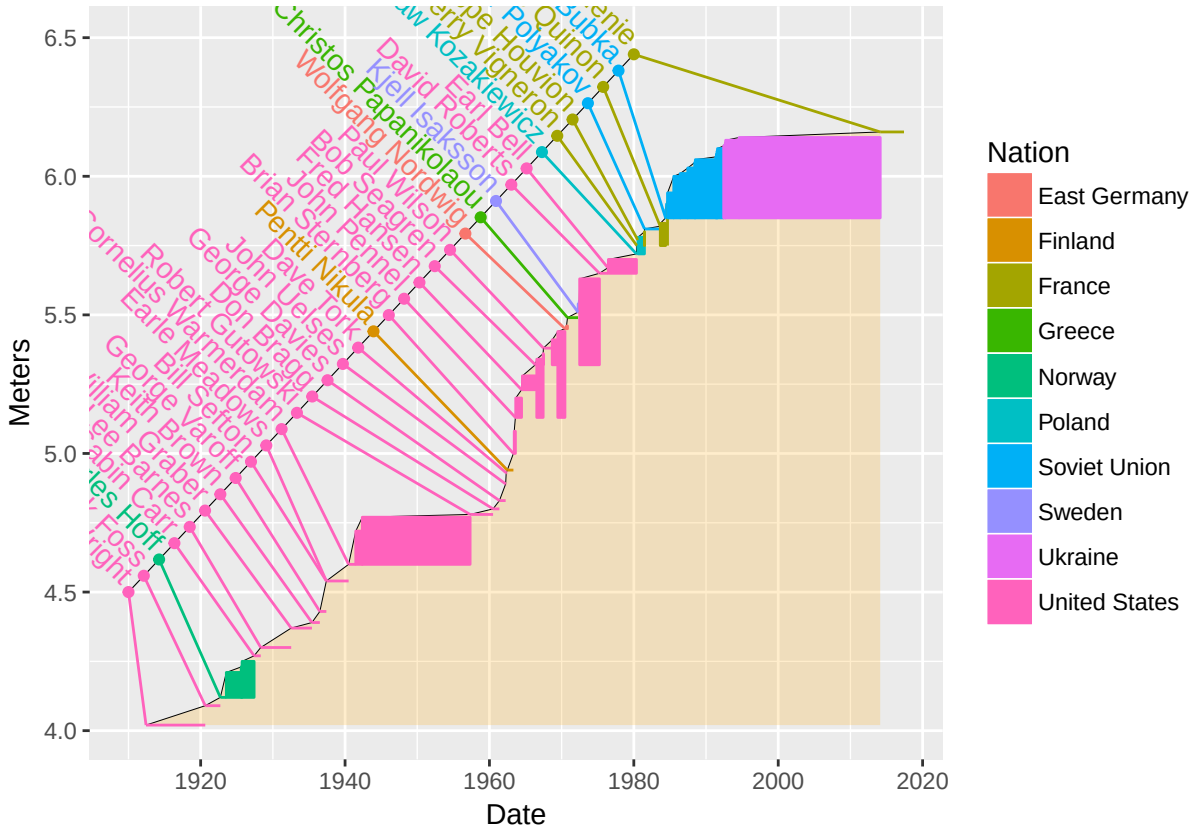
4. Now, little colored lines from the nameline points to the records. For this we can use the `geom_segment()` function.

```
bb3 <- bb2 + geom_segment(data = pv1, mapping = aes(xend = nameline.x, yend = nameline.y))
bb3
```



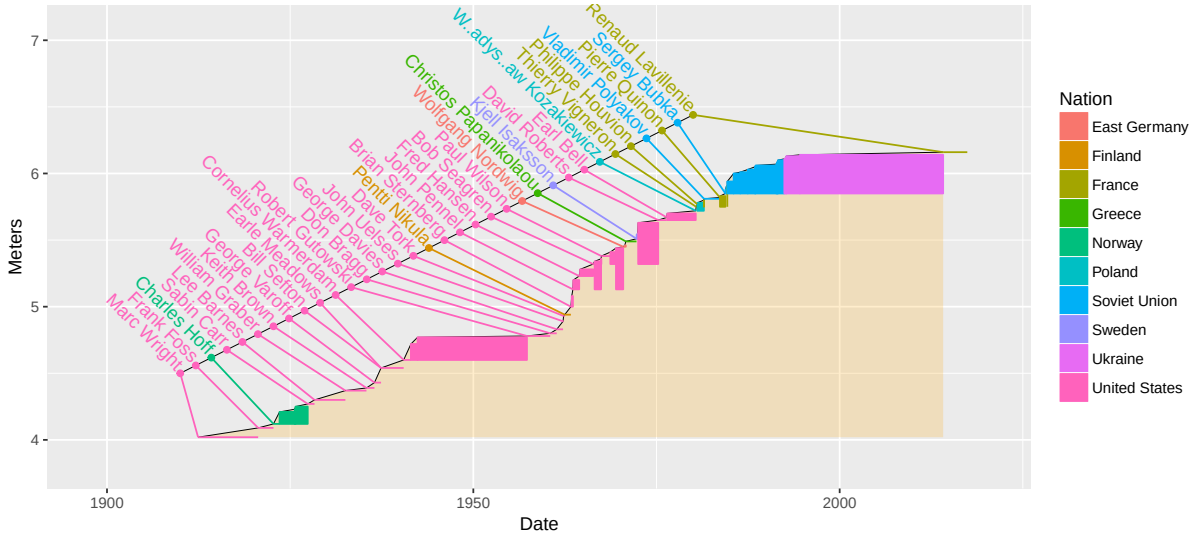
5. Finally, we want to add the names of the athletes in there. We already have their names in `pv1`. We use the `geom_text()` function.

```
bb4 <- bb3 + geom_text(data = pv1,
                       aes(label = Athlete,
                           x = nameline.x,
                           y = nameline.y + .05),
                       angle = -45, hjust = 1)
bb4
```



6. That is pretty cool, but a lot of names have gotten chopped off. Can we do something about that? There might be something a little more automatic, but we can do it by-hand, too:

```
bb4 + coord_cartesian(xlim = mdy(c("1-1-1898", "1-1-2020")), ylim = c(3.8, 7.1))
```



Chapter 8

Week 7: Making Simple Maps with R

Note that if you, the student, wish to run all the code yourself, you should download the `inputs` directory as a zipped file by going here with a web browser and then clicking the big "Download" button on the right. Once you have downloaded that, unzip it and put the whole `inputs` directory in the current working directory where you are working with R.

8.1 Intro

For a long time, R has had a relatively simple mechanism, via the `maps` package, for making simple outlines of maps and plotting lat-long points and paths on them.

More recently, with the advent of packages like `sp`, `rgdal`, and `rgeos`, R has been acquiring much of the functionality of traditional GIS packages (like ArcGIS, etc). This is an exciting development, and is now more easily accessible for the beginner than it was a few years ago when it required installation of specialized external libraries. Today, all the necessary libraries get downloaded when you install the needed packages from CRAN!

More recently, a third approach to convenient mapping, using `ggmap` has been developed that allows the tiling of detailed base maps from Google Earth or Open Street Maps, etc., upon which spatial data may be plotted.

Today, we are going to focus on mapping using base maps from R's tried and true `maps` package and also using the `ggmap` package. Next week we will cover some more advanced GIS-related topics using `rgdal`, or `sp` to plot maps with different projections, etc.

As in our previous explorations in this course, when it comes to plotting, we are going to completely skip over R's base graphics system and head directly to Hadley Wickham's `ggplot2` package. Hadley has included a few functions that make it relatively easy to interact with the data in R's `maps` package, and of course, once a map layer is laid down, you have all the power of ggplot at your fingertips to overlay whatever you may want to over the map.

`ggmap` is a package that goes out to different map servers and grabs base maps to plot things on, then it sets up the coordinate system and writes it out as the base layer for further ggplotting. It is pretty sweet, but does *not* support different projections.

8.1.1 Today's Goals

1. Introduce readers to the map outlines available in the `maps` package
    - Show how to convert those data into data frames that `ggplot2` can deal with
    - Discuss some `ggplot2` related issues about plotting things.

2. Use `ggmap` to make some pretty decent looking maps

I feel that the above two topics should cover a large part of what people will need for making useful maps of field sites, or sampling locations, or fishing track lines, etc.

For today we will be skipping how to read in traditional GIS "shapefiles" so as to minimize the number of packages that need installation, but keep in mind that it isn't too hard to do that in R, too. (We'll do that next week using the `ggspatial` package).

8.1.2 Prerequisites

You are going to need to install a few packages beyond the tidyverse.

```
# some standard map packages.
install.packages(c("maps", "mapdata"))

# the ggmap package.  Might as well get the bleeding edge version from GitHub
devtools::install_github("dkahle/ggmap")
```

8.1.3 Load up a few of the libraries we will use

```
library(tidyverse)
library(mapdata)
library(maps)
```

8.2 Plotting maps-package maps with ggplot

8.2.1 The main players:

- The `maps` package contains a lot of outlines of continents, countries, states, and counties that have been with R for a long time.
- The `mapdata` package contains a few more, higher-resolution outlines.
- The `maps` package comes with a plotting function, but, we will opt to use `ggplot2` to plot the maps in the `maps` package.
- Recall that `ggplot2` operates on data frames. Therefore we need some way to translate the `maps` data into a data frame format the `ggplot` can use. This is done for us with the lovely `map_data` function in 'ggplot2.

8.2.2 Maps in the maps package

- Package `maps` provides lots of different map outlines and points for cities, etc.
- Some examples: `usa`, `nz`, `state`, `world`, etc. Do `help(package = "maps")` to see more information.

8.2.3 Makin' data frames from map outlines

- `ggplot2` provides the `map_data()` function.

  - Think of it as a function that turns a series of points along an outline into a data frame of those points.
  - Syntax: `map_data("name")` where "name" is a quoted string of the name of a map in the `maps` or `mapdata` package

- Here we get a USA map from `maps`:

```
usa <- map_data("usa")

dim(usa)
```

```
## [1] 7243    6
```

```
head(usa)
```

```
##        long      lat group order region subregion
## 1 -101.4078 29.74224     1     1   main      <NA>
## 2 -101.3906 29.74224     1     2   main      <NA>
## 3 -101.3620 29.65056     1     3   main      <NA>
## 4 -101.3505 29.63911     1     4   main      <NA>
## 5 -101.3219 29.63338     1     5   main      <NA>
## 6 -101.3047 29.64484     1     6   main      <NA>
```

```
tail(usa)
```

```
##           long      lat group order         region subregion
## 7247 -122.6187 48.37482    10  7247 whidbey island      <NA>
## 7248 -122.6359 48.35764    10  7248 whidbey island      <NA>
## 7249 -122.6703 48.31180    10  7249 whidbey island      <NA>
## 7250 -122.7218 48.23732    10  7250 whidbey island      <NA>
## 7251 -122.7104 48.21440    10  7251 whidbey island      <NA>
## 7252 -122.6703 48.17429    10  7252 whidbey island      <NA>
```

- Here is the high-res world map centered on the Pacific Ocean from `mapdata`

```
w2hr <- map_data("world2Hires")

dim(w2hr)
```

```
## [1] 2274539       6
```

```
head(w2hr)
```

```
##       long      lat group order region subregion
## 1 226.6336 58.42416     1     1 Canada      <NA>
## 2 226.6314 58.42336     1     2 Canada      <NA>
## 3 226.6122 58.41196     1     3 Canada      <NA>
## 4 226.5911 58.40027     1     4 Canada      <NA>
## 5 226.5719 58.38864     1     5 Canada      <NA>
## 6 226.5528 58.37724     1     6 Canada      <NA>
```

```
tail(w2hr)
```

```
##             long      lat group   order      region subregion
## 2276817 125.0258 11.18471  2284 2276817 Philippines     Leyte
## 2276818 125.0172 11.17142  2284 2276818 Philippines     Leyte
## 2276819 125.0114 11.16110  2284 2276819 Philippines     Leyte
## 2276820 125.0100 11.15555  2284 2276820 Philippines     Leyte
## 2276821 125.0111 11.14861  2284 2276821 Philippines     Leyte
## 2276822 125.0155 11.13887  2284 2276822 Philippines     Leyte
```

8.2.4 The structure of those data frames

These are pretty straightforward:

- `long` is longitude. Things to the west of the prime meridian are negative, down to -180, and things to the east of the prime meridian run from 0 to positive 180.
- `lat` is latitude.
- `order`. This just shows in which order `ggplot` should "connect the dots"
- `region` and `subregion` tell what region or subregion a set of points surrounds.
- `group`. This is *very important*! `ggplot2`'s functions can take a group argument which controls (amongst other things) whether adjacent points should be connected by lines. If they are in the same group, then they get connected, but if they are in different groups then they don't.
  - Essentially, having two points in different groups means that `ggplot` "lifts the pen" when going between them.

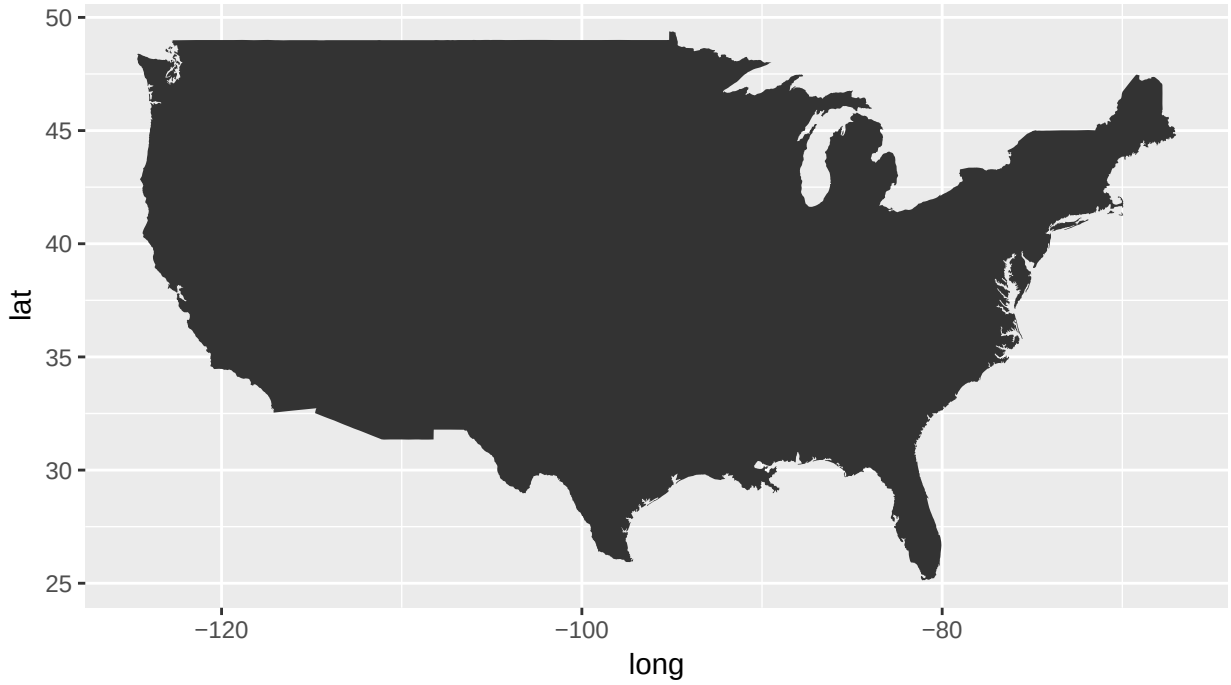
8.2.5 Plot the USA map

- Maps in this format can be plotted with the polygon geom. i.e. using `geom_polygon()`.
- `geom_polygon()` drawn lines between points and "closes them up" (i.e. draws a line from the last point back to the first point)
- You have to map the `group` aesthetic to the `group` column
- Of course, `x = long` and `y = lat` are the other aesthetics.

8.2.5.1 Simple black map

By default, `geom_polygon()` draws with no line color, but with a black fill:

```
usa <- map_data("usa") # we already did this, but we can do it again
ggplot() +
  geom_polygon(data = usa, aes(x = long, y = lat, group = group)) +
  coord_quickmap()
```



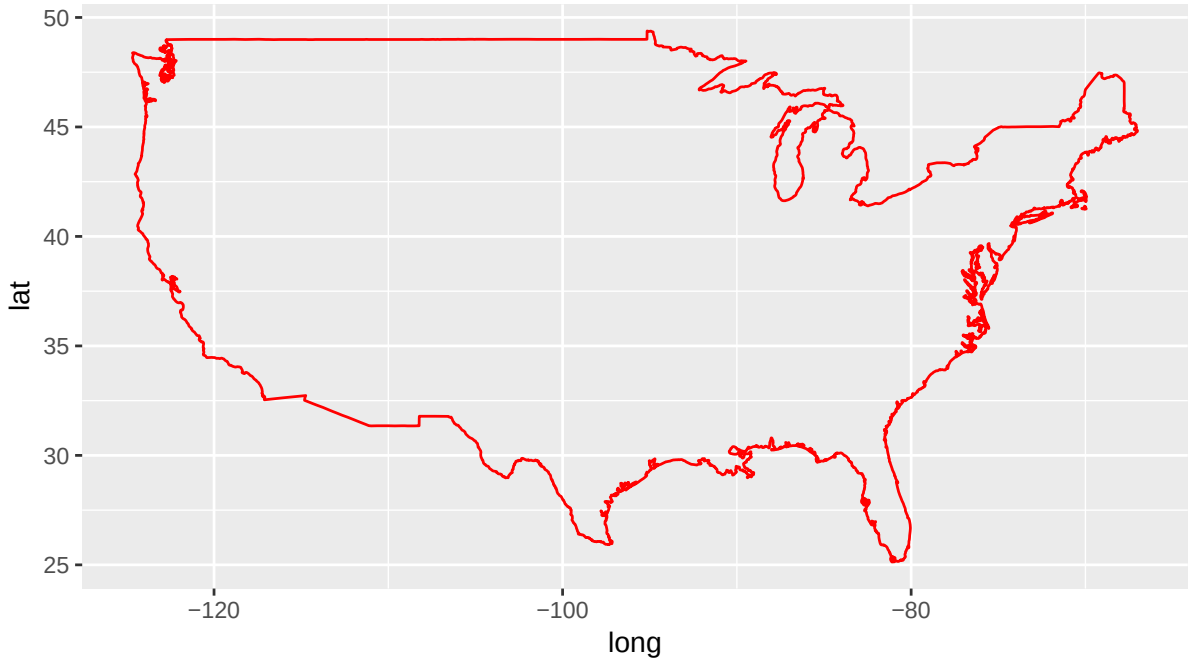
8.2.5.2 What is this coord_quickmap()?

- This is very important when drawing maps.
- It sets the relationship between one unit in the y direction and one unit in the x direction so that the *aspect ratio* is good for your map.
- Then, even if you change the outer dimensions of the plot (i.e. by changing the window size or the size of the pdf file you are saving it to (in `ggsave` for example)), the aspect ratio remains unchanged.

8.2.5.3 Mess with line and fill colors

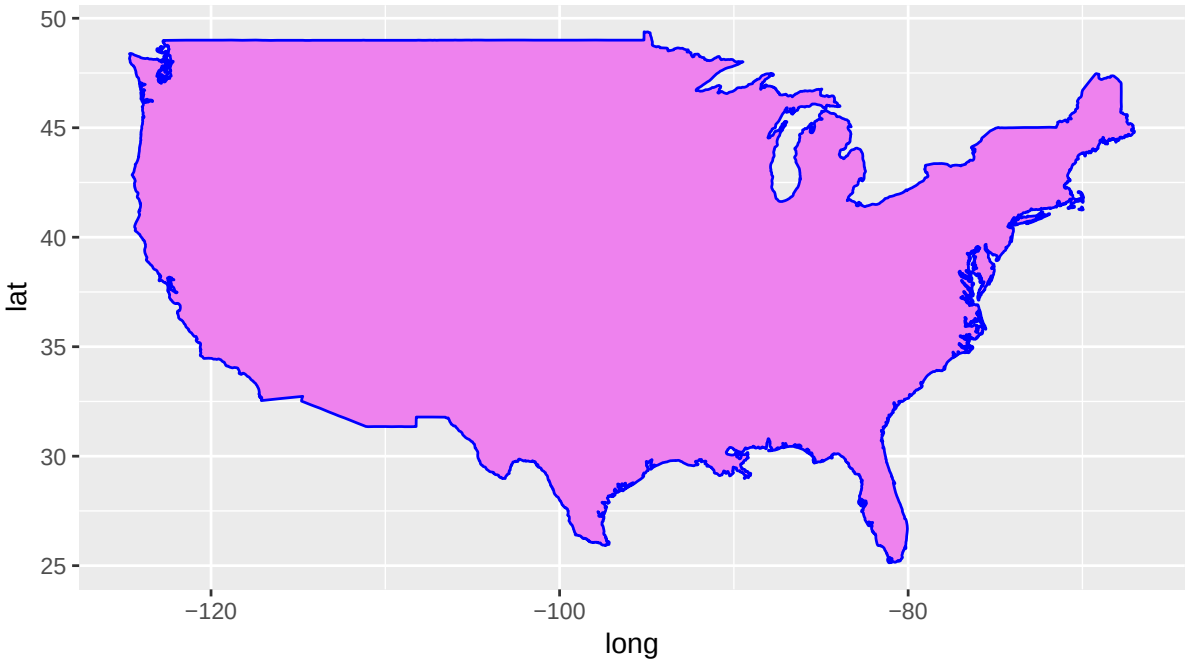
- Here is no fill, with a red line. Remember, fixed values of aesthetics (i.e., those that are not being mapped to a variable in the data frame) go *outside* the `aes` function.

```
ggplot() +
  geom_polygon(data = usa, aes(x = long, y = lat, group = group), fill = NA, color = "red") +
  coord_quickmap()
```



- Here is violet fill, with a blue line.

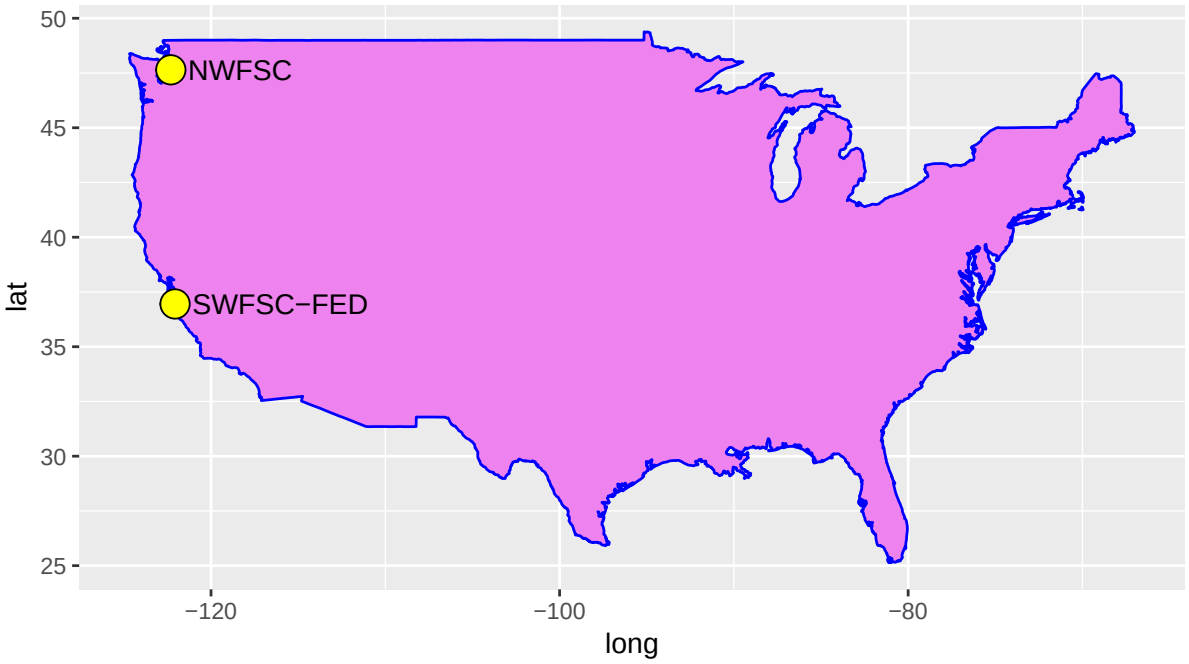
```
gg1 <- ggplot() +
  geom_polygon(data = usa, aes(x = long, y = lat, group = group), fill = "violet", color = "blue") +
  coord_quickmap()
gg1
```



8.2.5.4 Adding points to the map

- Let's add black and yellow points at the NMFS lab in Santa Cruz and at the Northwest Fisheries Science Center lab in Seattle.

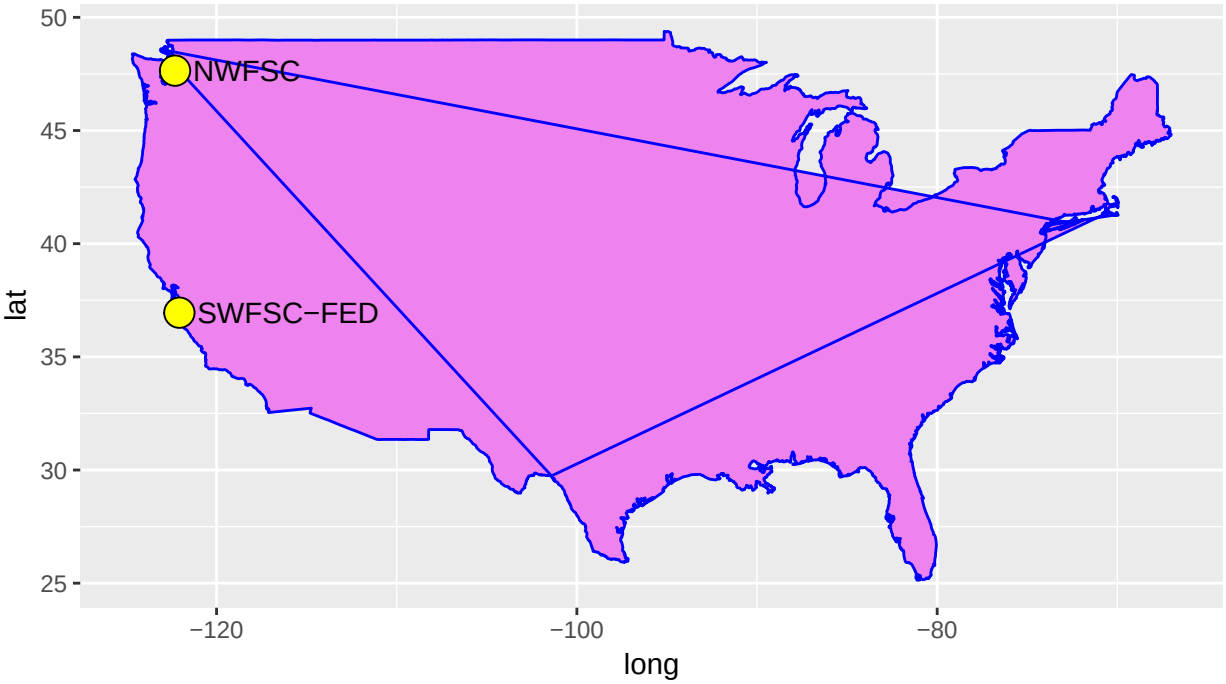
```
labs <- tibble(
  long = c(-122.064873, -122.306417),
  lat = c(36.951968, 47.644855),
  names = c("SWFSC-FED", "NWFSC"))

gg1 +
  geom_point(data = labs, aes(x = long, y = lat), shape = 21, color = "black", fill = "yellow", siz
  geom_text(data = labs, aes(x = long, y = lat, label = names), hjust = 0, nudge_x = 1)
```



8.2.5.5 See how important the group aesthetic is

Here we plot that map without using the group aesthetic:

```
ggplot() +
      geom_polygon(data = usa, aes(x = long, y = lat), fill = "violet", color = "blue") +
      geom_point(data = labs, aes(x = long, y = lat), shape = 21, color = "black", fill = "yellow", size
      geom_text(data = labs, aes(x = long, y = lat, label = names), hjust = 0, nudge_x = 1) +
      coord_quickmap()
```



That is no bueno! The lines are connecting points that should not be connected!

8.2.6 State maps

We can also get a data frame of polygons that tell us about state boundaries:

```
states <- map_data("state")
dim(states)
```

```
## [1] 15537     6
```

```
head(states)
```

```
##        long      lat group order  region subregion
## 1 -87.46201 30.38968     1     1 alabama      <NA>
## 2 -87.48493 30.37249     1     2 alabama      <NA>
## 3 -87.52503 30.37249     1     3 alabama      <NA>
## 4 -87.53076 30.33239     1     4 alabama      <NA>
## 5 -87.57087 30.32665     1     5 alabama      <NA>
## 6 -87.58806 30.32665     1     6 alabama      <NA>
```

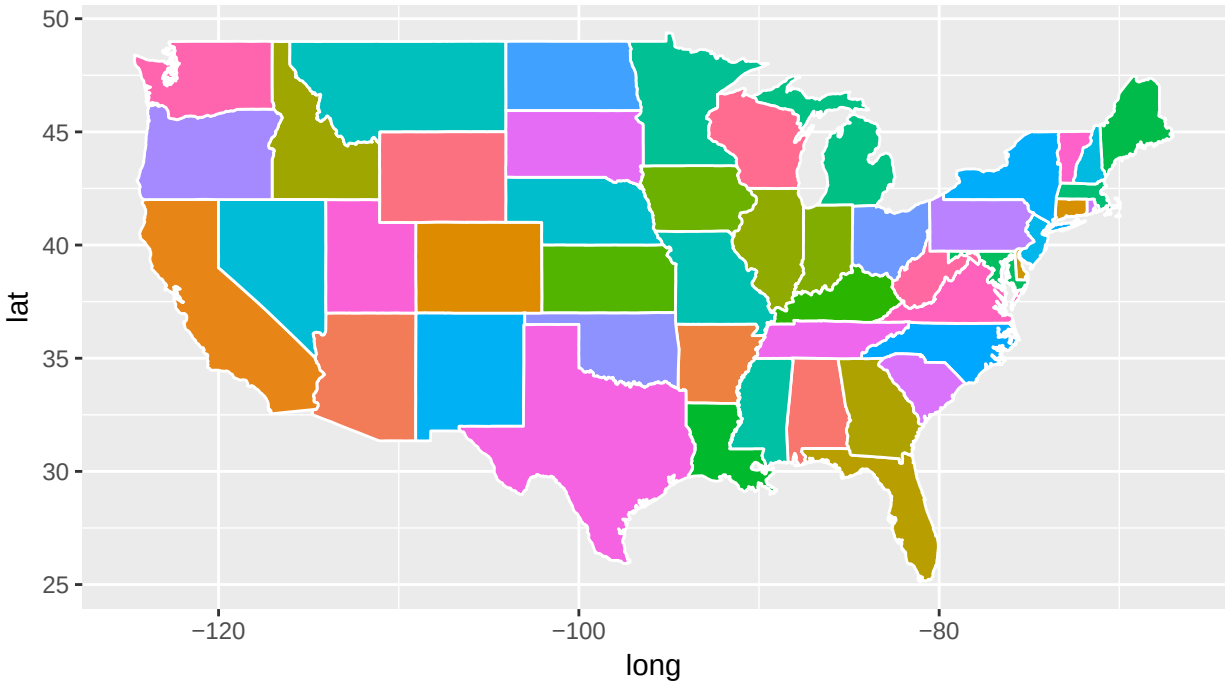
```
tail(states)
```

```
##            long      lat group order  region subregion
## 15594 -106.3295 41.00659    63 15594 wyoming      <NA>
## 15595 -106.8566 41.01232    63 15595 wyoming      <NA>
## 15596 -107.3093 41.01805    63 15596 wyoming      <NA>
## 15597 -107.9223 41.01805    63 15597 wyoming      <NA>
## 15598 -109.0568 40.98940    63 15598 wyoming      <NA>
## 15599 -109.0511 40.99513    63 15599 wyoming      <NA>
```

8.2.6.1 Plot all the states, all colored a little differently

This is just like it is above, but (using the `aes` function) we can map the `fill` aesthetic to `region` and make sure the the lines of state borders are white.

```
ggplot(data = states) +
  geom_polygon(aes(x = long, y = lat, fill = region, group = group), color = "white") +
  coord_quickmap() +
  guides(fill = FALSE)  # do this to leave off the color legend
```



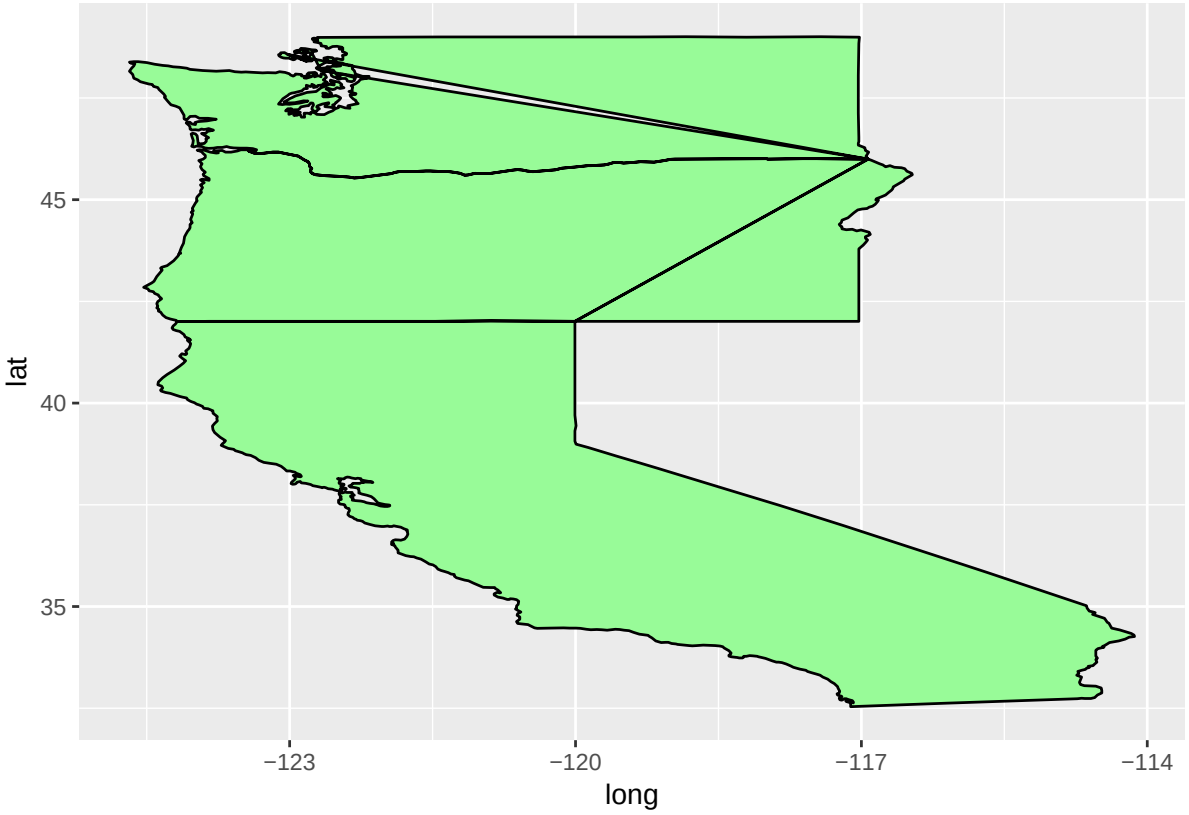
Boom! That is easy.

8.2.6.2 Plot just a subset of states in the contiguous 48:

- Because the `map_data()` sends back a data frame, we can use the tidy tools of `dplyr` to retain just certain parts of it.

- For example, we can grab just CA, OR, and WA and then plot them:

```
west_coast <- states %>%
  filter(region %in% c("california", "oregon", "washington"))

ggplot(data = west_coast) +
  geom_polygon(aes(x = long, y = lat), fill = "palegreen", color = "black")
```



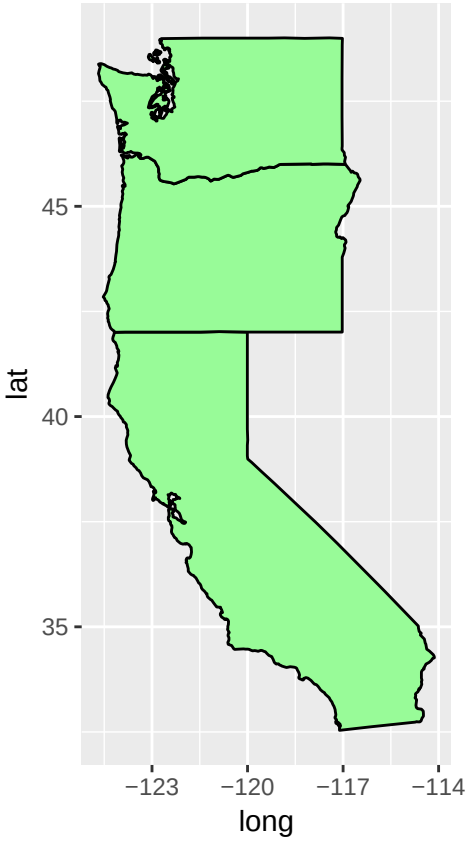
8.2.6.3 Man that is ugly!!

- I am just keeping people on their toes. What have we forgotten here?

  - `group`
  - `coord_quickmap()`

- Let's put those back in there:

```
ggplot(data = west_coast) +
  geom_polygon(aes(x = long, y = lat, group = group), fill = "palegreen", color = "black") +
  coord_quickmap()
```



Phew! That is a little better!

8.2.6.4 Zoom in on California and look at counties

- Getting the california data is easy:

```
ca_df <- states %>%
  filter(region == "california")

head(ca_df)
```

```
##        long      lat group order     region subregion
## 1 -120.0060 42.00927     4   667 california      <NA>
## 2 -120.0060 41.20139     4   668 california      <NA>
## 3 -120.0060 39.70024     4   669 california      <NA>
## 4 -119.9946 39.44241     4   670 california      <NA>
## 5 -120.0060 39.31636     4   671 california      <NA>
## 6 -120.0060 39.16166     4   672 california      <NA>
```

- Now, let's also get the county lines there

```
counties <- map_data("county")
ca_county <- counties %>%
  filter(region == "california")

head(ca_county)
```

```
##        long      lat group order     region subregion
## 1 -121.4785 37.48290   157  6965 california   alameda
```

```
## 2 -121.5129 37.48290   157  6966 california   alameda
## 3 -121.8853 37.48290   157  6967 california   alameda
## 4 -121.8968 37.46571   157  6968 california   alameda
## 5 -121.9254 37.45998   157  6969 california   alameda
## 6 -121.9483 37.47717   157  6970 california   alameda
```

- Plot the state first but let's ditch the axes gridlines, and gray background by using the super-wonderful `theme_void()` which leaves off everything except the geoms, and the guides if they are needed.

```
ca_base <- ggplot(data = ca_df, mapping = aes(x = long, y = lat, group = group)) +
  coord_quickmap() +
  geom_polygon(color = "black", fill = "gray")
ca_base + theme_void()
```

- Now plot the county boundaries in white:

```
ca_base + theme_void() +
  geom_polygon(data = ca_county, fill = NA, color = "white") +
  geom_polygon(color = "black", fill = NA)  # get the state border back on top
```

8.2.6.5 Get some facts about the counties

- The above is pretty cool, but it seems like it would be a lot cooler if we could plot some information about those counties.

- Now I can go to wikipedia or http://www.california-demographics.com/counties_by_population and grab population and area data for each county.
- In fact, I copied their little table on Wikipedia and saved it into `inputs/ca-counties-wikipedia.txt`. In full disclosure I also edited the name of San Francisco from "City and County of San Francisco" to "San Francisco County" to be like the others (and not break my regular expression searches)
- 2016 Edit! I had originally pumped the matrix resulting from the `str_match` below through a few other `stringr` function (like `str_replace_all`) which, apparently, in an older version of `stringr` maintained the matrix form, but in a newer version, squashed everything into a vector. This was kindly noted by hueykwik in this issue. So the code below is modified from what it used to be, but seems to work now. Just as an aside, in the years since I first put this course together (in 2014), things have changed quite a bit in the data analysis world, and, as we are doing in this course again in 2017, we are spending almost all our time in the tidyverse.
- Watch this regex fun. Note that if you, the student, wish to run all the code yourself, you should download the `inputs` directory as a zipped file by going here with a web browser and then clicking the big "Download" button on the right. Once you have downloaded that, unzip it and put the whole `inputs` directory in the current working directory where you are working with R.

```
library(stringr)
library(dplyr)

# make a data frame
x <- readLines("inputs/ca-counties-wikipedia.txt")
pop_and_area <- str_match(x, "^([a-zA-Z ]+)County\t.*\t([0-9,]{2,10})\t([0-9,]{2,10}) sq mi$")[, -1]
```

```
      na.omit() %>%
      as.data.frame(stringsAsFactors = FALSE) %>%
      mutate(subregion = str_trim(V1) %>% tolower(),
             population = as.numeric(str_replace_all(V2, ",", "")),
             area = as.numeric(str_replace_all(V3, ",", ""))
      ) %>%
      select(subregion, population, area) %>%
      tbl_df()

    head(pop_and_area)
```

```
## # A tibble: 6 × 3
##   subregion population  area
##       <chr>      <dbl> <dbl>
## 1   alameda    1578891   738
## 2    alpine       1159   739
## 3    amador      36519   593
## 4     butte     222090  1640
## 5 calaveras      44515  1020
## 6    colusa      21358  1151
```

- We now have the numbers that we want, but we need to attach those to every point on polygons of the counties. This is, of course, a job for `left_join` from the `dplyr` package, and while we are at it, we will add a column of `people_per_mile`

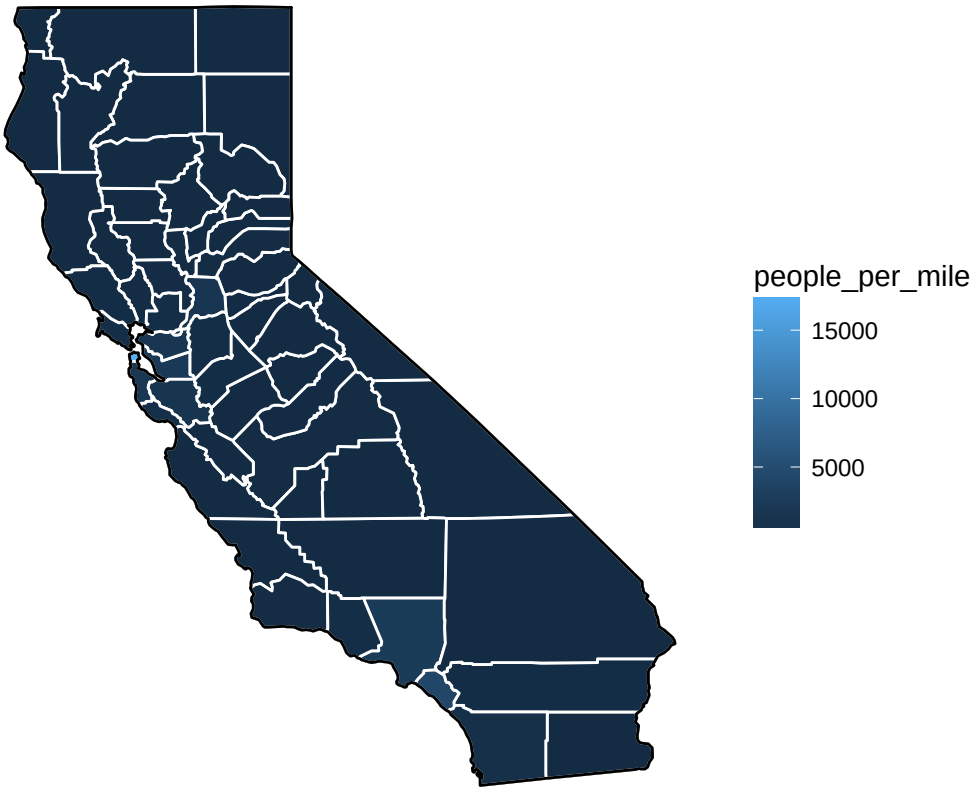
```
cacopa <- left_join(ca_county, pop_and_area, by = "subregion") %>%
  mutate(people_per_mile = population / area)
head(cacopa)
```

```
##        long      lat group order     region subregion population area
## 1 -121.4785 37.48290   157  6965 california   alameda    1578891  738
## 2 -121.5129 37.48290   157  6966 california   alameda    1578891  738
## 3 -121.8853 37.48290   157  6967 california   alameda    1578891  738
## 4 -121.8968 37.46571   157  6968 california   alameda    1578891  738
## 5 -121.9254 37.45998   157  6969 california   alameda    1578891  738
## 6 -121.9483 37.47717   157  6970 california   alameda    1578891  738
##   people_per_mile
## 1        2139.419
## 2        2139.419
## 3        2139.419
## 4        2139.419
## 5        2139.419
## 6        2139.419
```

8.2.6.6 Now plot population density by county

If you were needing a little more elbow room in the great Golden State, this shows you where you can find it:

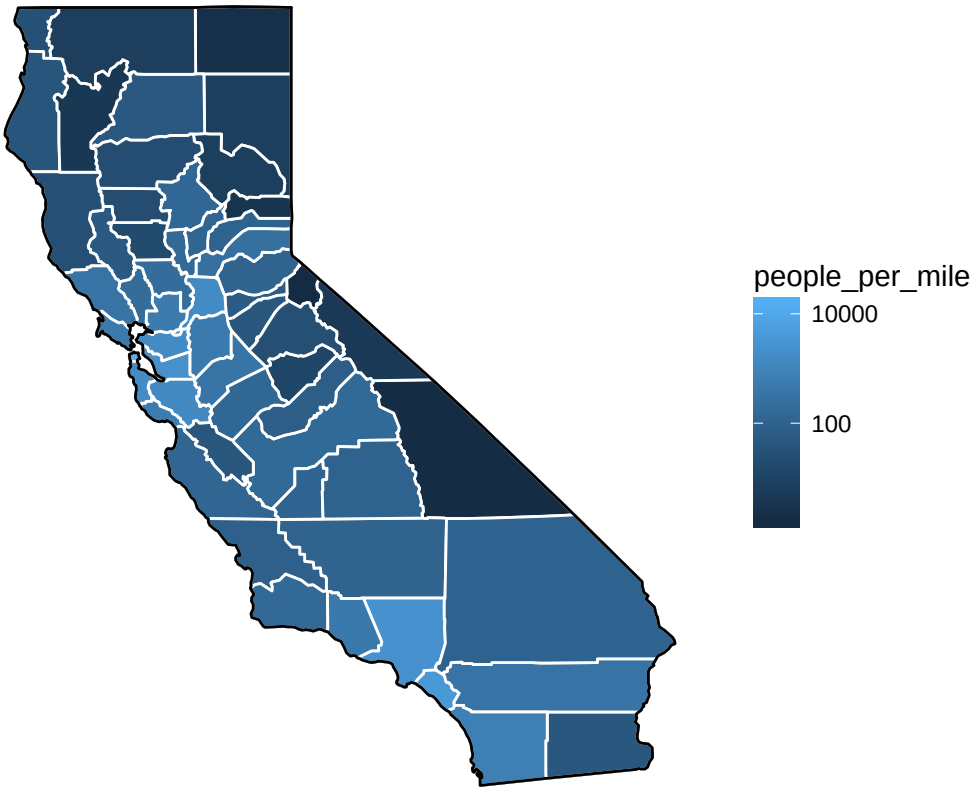
```
elbow_room1 <- ca_base +
      geom_polygon(data = cacopa, aes(fill = people_per_mile), color = "white") +
      geom_polygon(color = "black", fill = NA) +
      theme_void()

elbow_room1
```



8.2.6.7 Lame!

- The popuation density in San Francisco is so great that it makes it hard to discern differences between other areas.
- This is a job for a scale transformation. Let's take the log-base-10 of the population density.
- Instead of making a new column which is log10 of the `people_per_mile` we can just apply the transformation in the gradient using the `trans` argument

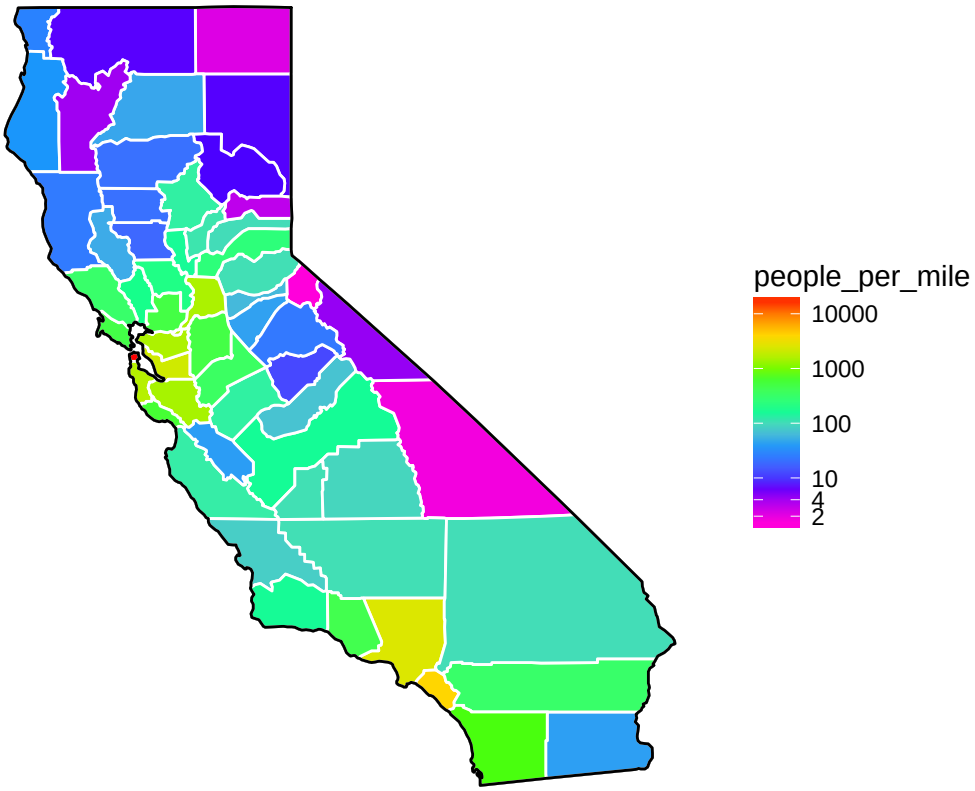
```
elbow_room1 + scale_fill_gradient(trans = "log10")
```



8.2.6.8 Still not great

I personally like more color than `ggplot` uses in its default gradient. In that respect I gravitate more toward Matlab's default color gradient. Can we do something similar with `ggplot`? (It turns out we can…my apologies to those purists who would tell me that rainbow colors are not good for showing quantitative scales—I think that in some cases they work reasonaly well…for example the following…of course, there are times when they work abysmally, as well.)

```
eb2 <- elbow_room1 +
    scale_fill_gradientn(colours = rev(rainbow(7)),
                         breaks = c(2, 4, 10, 100, 1000, 10000),
                         trans = "log10")
eb2
```



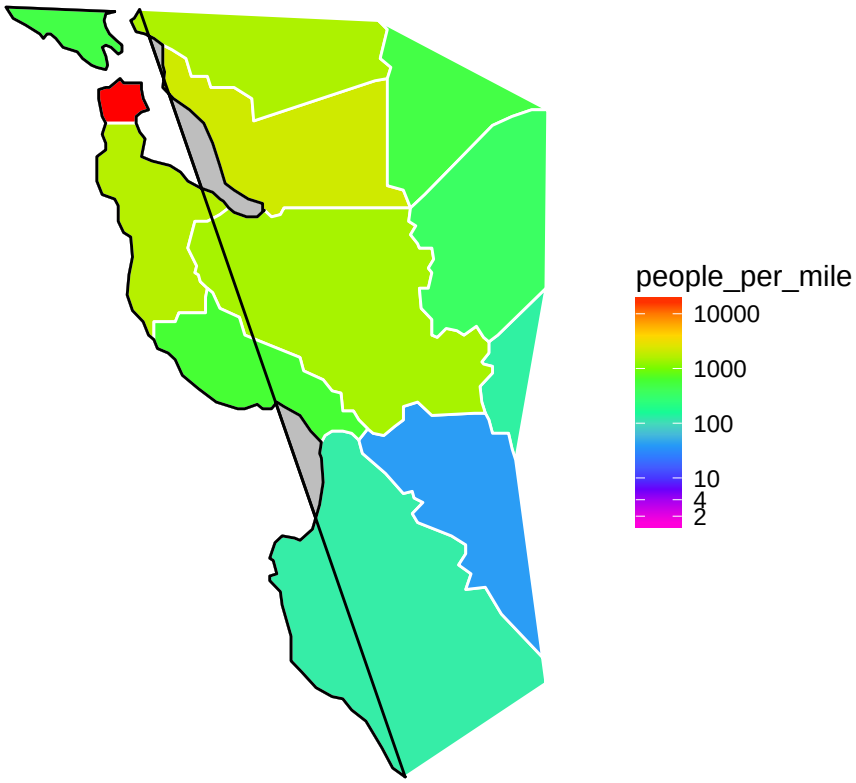
That is reasonably cool.

8.2.7 zoom in?

Note that the scale of these maps from package `maps` are not great. We can zoom in to the Bay region, and it sort of works scale-wise, but if we wanted to zoom in more, it would be tough.

Let's try!

```
eb2 + xlim(-123, -121.0) + ylim(36, 38)
```



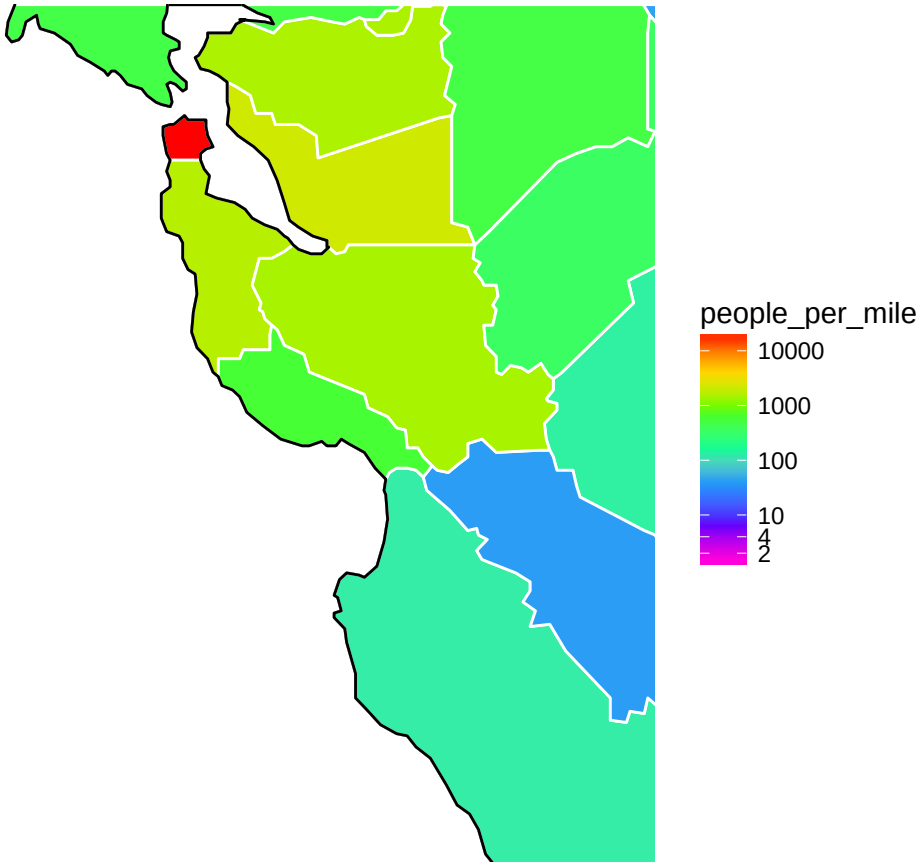
- Whoa! That is an epic fail. Why?
- Recall that `geom_polygon()` connects the end point of a `group` to its starting point.
- And the kicker: the `xlim` and `ylim` functions in `ggplot2` discard all the data that is not within the plot area.
  - Hence there are new starting points and ending points for some groups (or in this case the black-line permiter of California) and those points get connected. Not good.

8.2.8 True zoom.

- If you want to keep all the data the same but just zoom in, you can use the `xlim` and `ylim` arguments to `coord_cartesian()`. Though, to keep the aspect ratio correct we must use `coord_quickmap()` instead of `coord_cartesian()`.

- This chops stuff off but doesn't discard it from the data set:

```
eb2 + coord_quickmap(xlim = c(-123, -121.0),  ylim = c(36, 38))
```



Side Note: The coastline in the `maps` package is pretty low-resolution, and looks like a heap of bird droppings when you zoom way in on it.

If you want to get much more accurate coastlines, you can use better data sources like NOAA's GSHHS: A Global Self-consistent, Hierarchical, High-resolution Geography Database. I have a blog post about that at http://eriqande.github.io/2014/12/17/compare-map-resolutions.html.

8.3 ggmap

The `ggmap` package is a nice way to make quick maps with decent backgrounds! We will talk next week about how to get better-looking maps at some resolutions by using shapefiles and rasters from naturalearthdata.com but `ggmap` will get you 95% of the way there with a lot less of the work!

If you don't have it, get the most updated version off GitHub:

```
devtools::install_github("dkahle/ggmap")
```

And load it like so:

```
library(ggmap)
```

Note, if you use it in a paper, please cite it:

```
citation("ggmap")
```

```
##
## To cite ggmap in publications, please use:
##
##   D. Kahle and H. Wickham. ggmap: Spatial Visualization with
```

```
##   ggplot2. The R Journal, 5(1), 144-161. URL
##   http://journal.r-project.org/archive/2013-1/kahle-wickham.pdf
##
## A BibTeX entry for LaTeX users is
##
##   @Article{,
##     author = {David Kahle and Hadley Wickham},
##     title = {ggmap: Spatial Visualization with ggplot2},
##     journal = {The R Journal},
##     year = {2013},
##     volume = {5},
##     number = {1},
##     pages = {144--161},
##     url = {http://journal.r-project.org/archive/2013-1/kahle-wickham.pdf},
##   }
```

8.3.1 Three examples

- I am going to run through three examples. Working from the small spatial scale up to a larger spatial scale.
  1. Named "sampling" points on the Sisquoc River from the "Sisquoctober Adventure"
  2. A GPS track from a short bike ride in Wilder Ranch.
  3. Fish sampling locations from the coded wire tag data base.

8.3.2 How ggmap works

- ggmap simplifies the process of downloading base maps from [Google Maps(https://www.google.com/maps) or Open Street Map or Stamen Maps to use in the background of your plots.
- It also sets the axis scales, etc, in a nice way.

- Once you have gotten your maps, you make a call with `ggmap()` much as you would with `ggplot()`
- Let's do by example.

8.3.3 Sisquoctober

- Here is a small data frame of points from the Sisquoc River.

```
sisquoc <- read.table("inputs/sisquoc-points.txt", sep = "\t", header = TRUE)
sisquoc
```

```
##     name       lon      lat
## 1    a17 -119.7603 34.75474
## 2 a20-24 -119.7563 34.75380
## 3 a25-28 -119.7537 34.75371
## 4 a18,19 -119.7573 34.75409
## 5 a35,36 -119.7467 34.75144
## 6    a31 -119.7478 34.75234
## 7    a38 -119.7447 34.75230
## 8    a43 -119.7437 34.75251
```

```
# note that ggmap tends to use "lon" instead of "long" for longitude.
```

You can grab Google map tiles by specifying a center and a zoom value that can be between 3 and 20 inclusive. Google map tiles are always squarish tiles. A zoom level of 3 is "world scale", while a zoom value

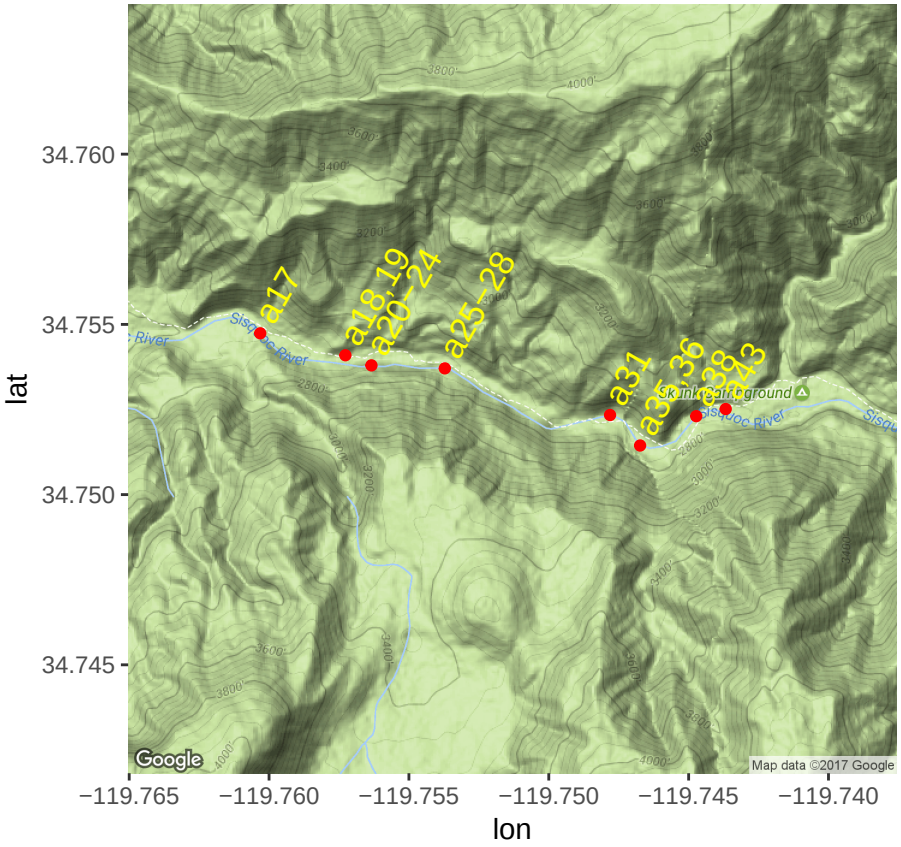
of 20 is "house scale." Our points in the Sisquoc are at a neighborhood scale, so we will try `zoom = 15`. Truth be told it takes some fiddling to get the zoom right.

8.3.3.1 Google's "Terrain-map" style

This is a nice choice for natural areas where you want to see some topography:

```
get_googlemap(center = c(mean(sisquoc$lon), mean(sisquoc$lat)), zoom = 15) %>%
  ggmap() +
  geom_point(data = sisquoc, mapping = aes(x = lon, y = lat), color = "red") +
  geom_text(data = sisquoc, aes(label = name), angle = 60, hjust = 0,
            color = "yellow", nudge_x = .0001, nudge_y = .0004, size = 4.5)
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=34.753117,-119.751324&zoom=15&size=640
```

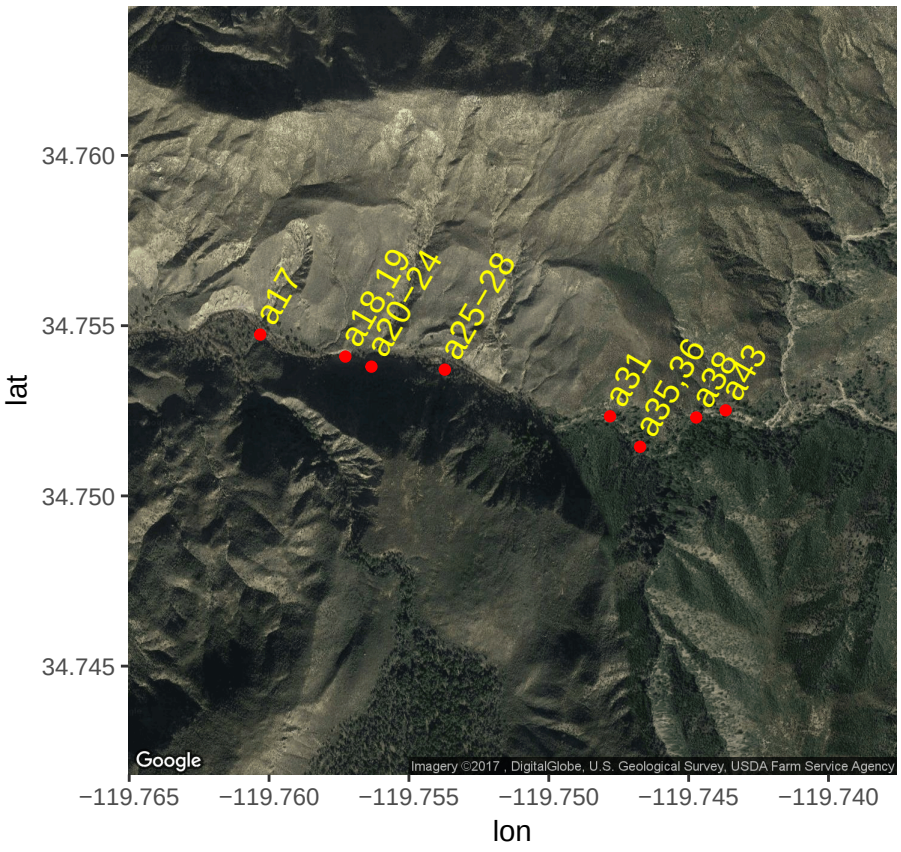


8.3.3.2 Google's "Satellite" maptype

This can be nice too...

```
get_googlemap(center = c(mean(sisquoc$lon), mean(sisquoc$lat)),
              zoom = 15,
              maptype = "satellite") %>%
  ggmap() +
  geom_point(data = sisquoc, mapping = aes(x = lon, y = lat), color = "red") +
  geom_text(data = sisquoc, aes(label = name), angle = 60, hjust = 0,
            color = "yellow", nudge_x = .0001, nudge_y = .0004, size = 4.5)
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=34.753117,-119.751324&zoom=15&size=640
```

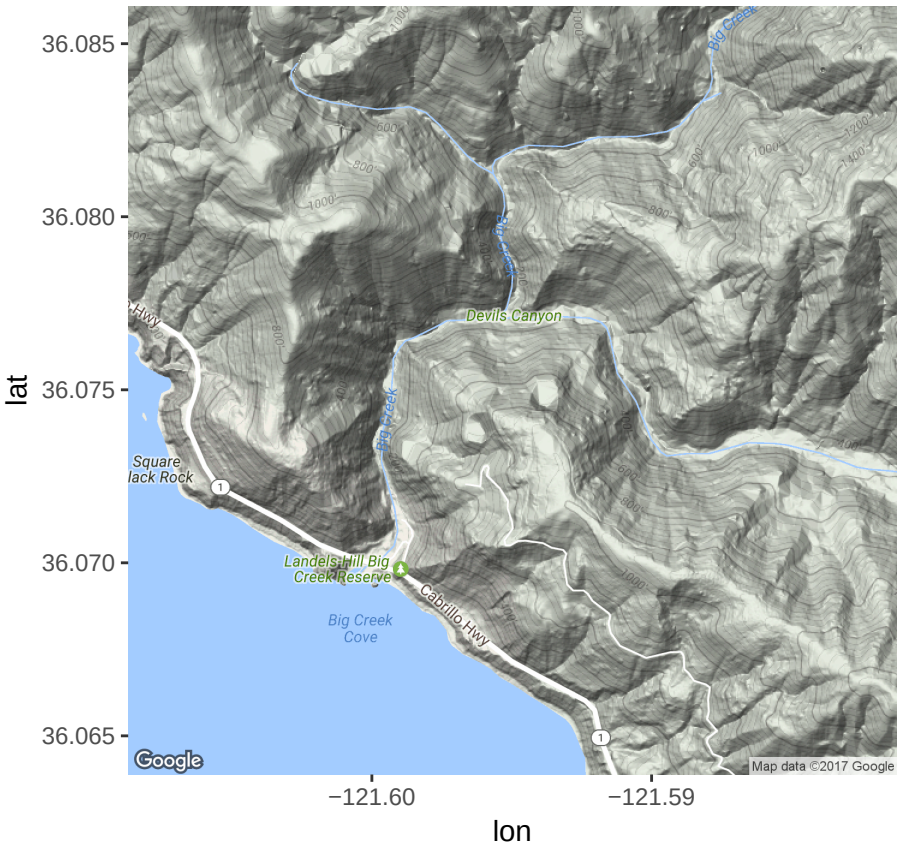


8.3.4 Big Creek in Big Sur

I have been involved in some work at Big Creek, and would like to replace one of our maps with something better. So, out of curiosity, let's see what Big Creek looks like with Google Maps.

```
get_googlemap(center = c(-121.595, 36.075), zoom = 15, maptype = "terrain") %>%
  ggmap()
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=36.075,-121.595&zoom=15&size=640x640&s
```



Hmm...that is quite nice.

8.3.4.1 Stamen maps

Can we get the stamen maps to work? Yes, but, for field sites and such, they don't seem nearly as useful as Google Maps. Here is some code but we don't bother running it because it looks really bad anyway.

```
bounds <- c(left = -119.77,
            bottom = 34.750,
            right = -119.73,
            top = 34.755)

# note, if you get these in the wrong order you get a totally
# uninformative error message.

sisquoc_stamen <- get_stamenmap(bbox = bounds, maptype = "terrain", zoom = 16)

# this looks like crap!

# maybe at bigger zoom levels it would be OK.
```

8.3.5 How about a bike ride?

- I was riding my bike one day with my phone and downloaded the GPS readings at short intervals.
- We can plot the route like this:

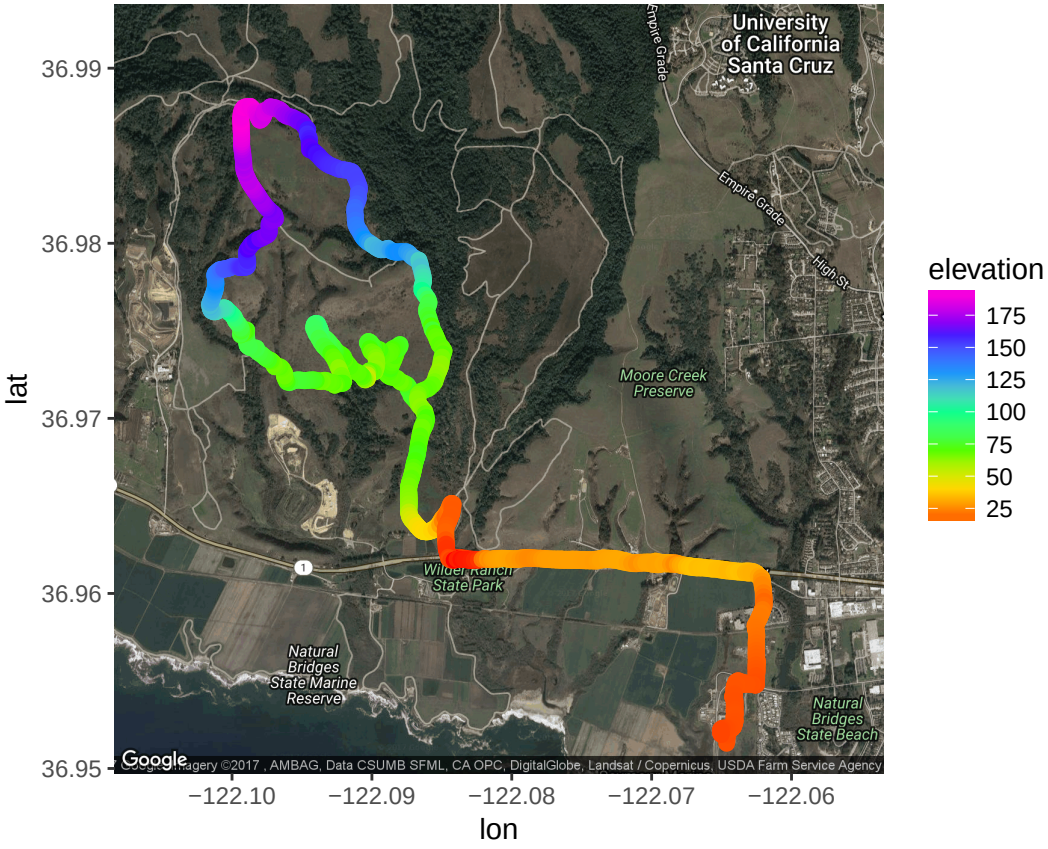
```
bike <- read.csv("inputs/bike-ride.csv")
head(bike)
```

```
##         lon      lat elevation                 time
## 1 -122.0646 36.95144      15.8 2011-12-08T19:37:56Z
## 2 -122.0646 36.95191      15.5 2011-12-08T19:37:59Z
## 3 -122.0645 36.95201      15.4 2011-12-08T19:38:04Z
## 4 -122.0645 36.95218      15.5 2011-12-08T19:38:07Z
## 5 -122.0643 36.95224      15.7 2011-12-08T19:38:10Z
## 6 -122.0642 36.95233      15.8 2011-12-08T19:38:13Z
```

```
bikemap1 <- get_googlemap(center = c(-122.080954, 36.971709), zoom = 14, maptype = "hybrid")
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=36.971709,-122.080954&zoom=14&size
```

```
ggmap(bikemap1) +
  geom_path(data = bike, aes(colour = elevation), size = 3, lineend = "round") +
  scale_color_gradientn(colours = rainbow(7), breaks = seq(25, 200, by = 25))
```



- See how we have mapped elevation to the color of the path using our rainbow colors again.
- Note that getting the right zoom and position for the map is sort of trial and error. You can go to google maps to figure out where the center should be: double click a location to get the lat-long of any spot.
- There is a `make_bbox` function that is supposed to expedite figuring out the zoom, but it has never really worked well for me with Google maps at large zooms. We will see it in action, below, however.

8.3.6 Fish sampling locations

For this, I have whittled down some stuff in the coded wire tag data base to georeferenced marine locations in British Columbia where at least one Chinook salmon was recovered between 2000 and 2012 inclusive. To see how I did all that you can check out this

Let's have a look at the data:

```
bc <- readRDS("inputs/bc_sites.rds")

# look at some of it:
bc %>%
  select(state_or_province:sub_location, longitude, latitude)
```

```
## # A tibble: 1,113 × 9
##    state_or_province water_type sector region  area location sub_location
##                <chr>      <chr>  <chr>  <chr> <chr>    <chr>        <chr>
## 1                  2          M      S     22  016   THOR IS           01
## 2                  2          M      N     26  012   MITC BY           18
## 3                  2          M      S     22  015   HARW IS           02
## 4                  2          M      N     26  006   HOPK PT           01
## 5                  2          M      S     23  017   TENT IS           06
## 6                  2          M      S     28  23A   NAHM BY           02
## 7                  2          M      N     26  006   GIL IS            06
## 8                  2          M      S     27  024   CLEL IS           06
## 9                  2          M      S     27  23B   SAND IS           04
## 10                 2          M      N     26  012   DUVA IS           16
## # ... with 1,103 more rows, and 2 more variables: longitude <dbl>,
## #   latitude <dbl>
```

So, we have 1,113 points to play with.

8.3.6.1 What do we hope to learn?

- These locations in BC are hierarchically structured. I am basically interested in how close together sites in the same "region" or "area" or "sector" are, and pondering whether it is OK to aggregate fish recoveries at a certain level for the purposes of getting a better overall estimate of the proportion of fish from different hatcheries in these areas.
- So, pretty simple stuff. I just want to plot these points on a map, and paint them a different color according to their sector, region, area, etc.
- Let's just enumerate things first, using `dplyr`:

```
bc %>%
  count(sector, region, area)
```

```
## Source: local data frame [42 x 4]
## Groups: sector, region [?]
##
##    sector region  area     n
##     <chr>  <chr> <chr> <int>
## 1              48  008      1
## 2              48  028      1
## 3              48  311      1
## 4       N      25  001     33
## 5       N      25  003     15
```

```
## 6       N     25  004     44
## 7       N     25  02E      2
## 8       N     25  02W     34
## 9       N     26  006     28
## 10      N     26  007     23
## # ... with 32 more rows
```

```
bc %>%
  count(sector, region)
```

```
## Source: local data frame [11 x 3]
## Groups: sector [?]
##
##    sector region     n
##     <chr>  <chr> <int>
## 1              48     3
## 2       N     25   128
## 3       N     26   174
## 4       S     22   191
## 5       S     23   136
## 6       S     24    93
## 7       S     27   247
## 8       S     28    23
## 9       S     61    49
## 10      S     62    66
## 11      S     AF     3
```

- That looks good. It appears like we could probably color code over the whole area down to region, and then down to area within subregions.

8.3.6.2 Makin' a map.

- Let us try to use `make_bbox()` to see if it will work better when used on a large scale. Note: in order to use an approximate bounding box, we have to use `get_map()` rather than `get_googlemap()`. Doing so looks like this…

```
# compute the bounding box
bc_bbox <- make_bbox(lat = latitude, lon = longitude, data = bc)
bc_bbox
```

```
##       left     bottom      right        top
## -133.63297   47.92497 -122.33652   55.80833
```

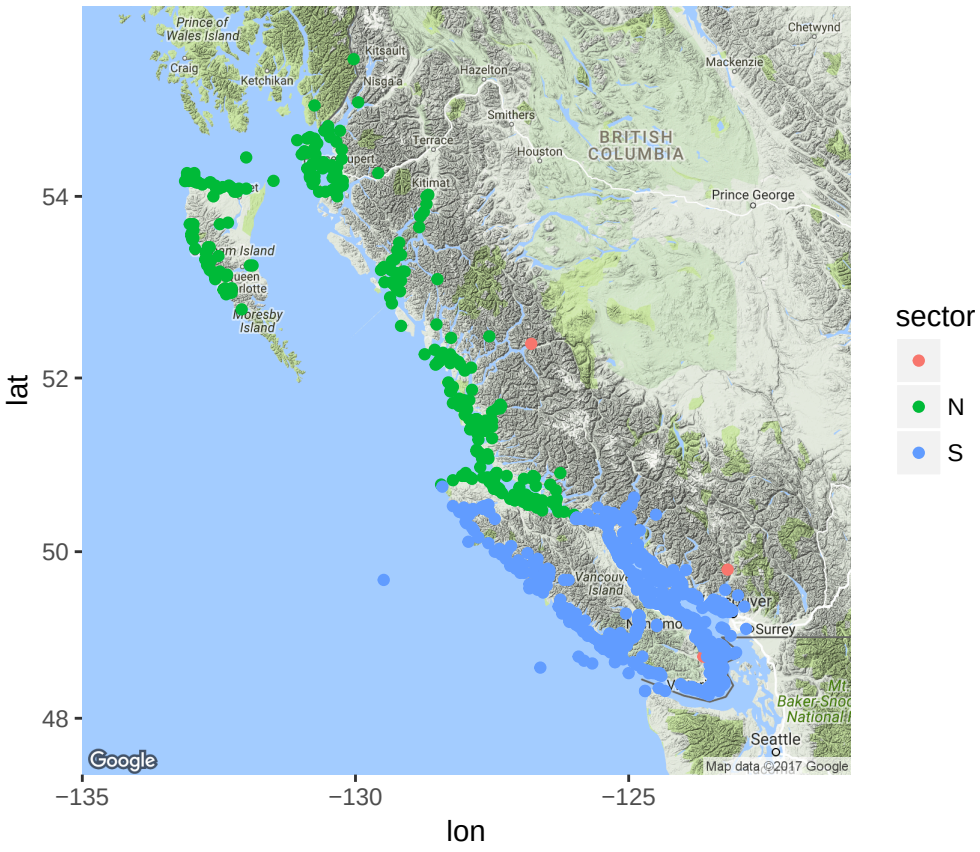
```
# grab the maps from google
bc_big <- get_map(location = bc_bbox, maptype = "terrain", source = "google")
```

```
## Warning: bounding box given to google - spatial extent only approximate.
```

```
## converting bounding box to center/zoom specification. (experimental)
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=51.86665,-127.98475&zoom=6&size=640x64
```

```
# plot the points and color them by sector
ggmap(bc_big) +
  geom_point(data = bc, mapping = aes(x = longitude, y = latitude, color = sector))
```

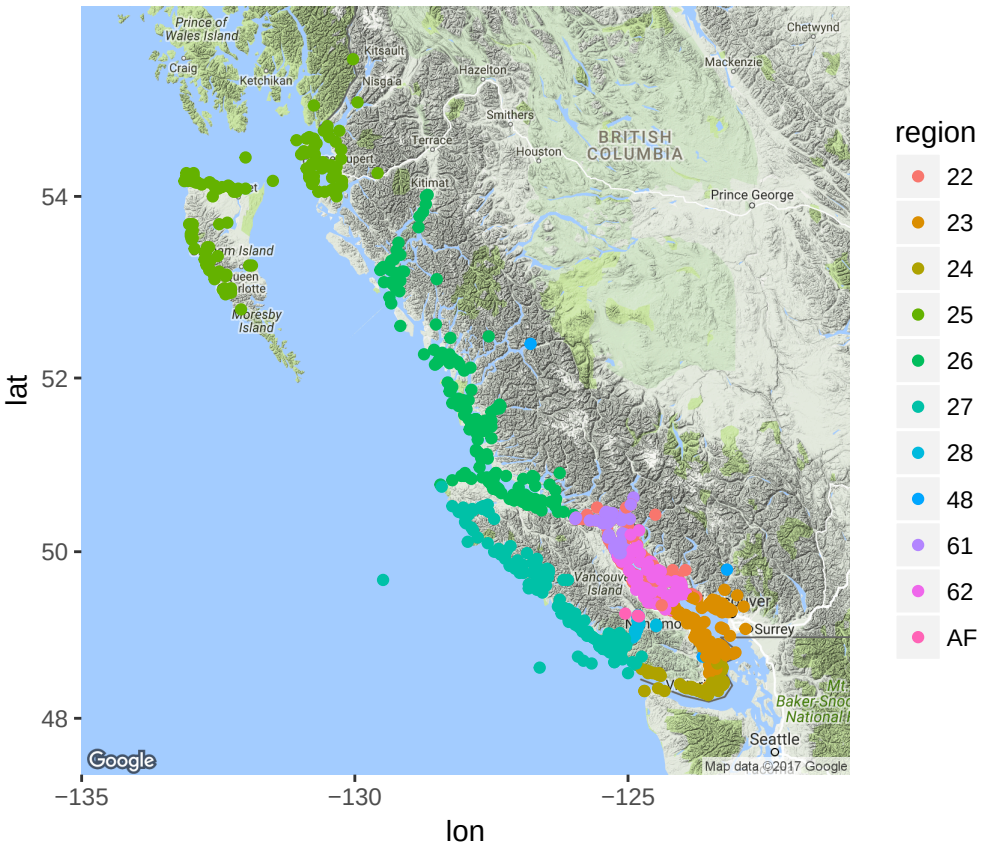


- Cool! That was about as easy as could be. North is in the north, south is in the south, and the three reddish points are clearly aberrant ones at the mouths of rivers.

8.3.6.3 Coloring it by region

- We should be able to color these all by region to some extent (it might get overwhelming), but let us have a go with it.
- Notice that region names are unique overall (not just within N or S) so we can just color by region name.

```
ggmap(bc_big) +
  geom_point(data = bc, mapping = aes(x = longitude, y = latitude, color = region))
```



- Once again that was dirt easy, though at this scale with all the different regions, it is hard to resolve all the colors. In general, it gets hard to resolve more than 7 or 8 colors on a map.

8.4 In-class review and short assignment

A lot of stuff went by in the last few sections. Here are the key players, in review:

Using the `maps` package with `ggplot2`:

- `library(maps)` this loads the library that has the map data.
- `map_data()` this function from `ggplot2` grabs map data from the `maps` package and returns it as a data frame `ggplot` can handle. Choices for the arguments are many. Some common ones are `"states"`, `"counties"`, `"usa"`, etc.
- `geom_polygon()` plots boundaries from the maps package.
- Don't forget: `group = group` in the aesthetics!
- `coord_quickmap()` ggplot function to get a good aspect ratio for a lat-long plot. (Note that if you want to do projections, with `maps` data you can use `coord_map()`…next week, maybe.)
- Zoom in without improper chopping with `coord_quickmap(xlim = ..., ylim = ...)`

Using `ggmap`:

- Get the latest version from GitHub: `devtools::install_github("dkahle/ggmap")`
- `get_googlemap(center = c(long, lat), zoom = 3 to 20)`
- Put the output of `get_googlemap()` into the `ggmap()` function and then proceed as you would with ggplot.
- Google Maps maptypes: "terrain", "satellite", "hybrid"

8.4.1 Short assignment of plotting with `maps`

If you don't have your own data to start plotting, you can try to make the following plot of the four "four-corners" states and their capitals. I have a list of capitals in the course repository that I read in like this:

```
caps <- read_csv("inputs/state_capitals.csv")
```

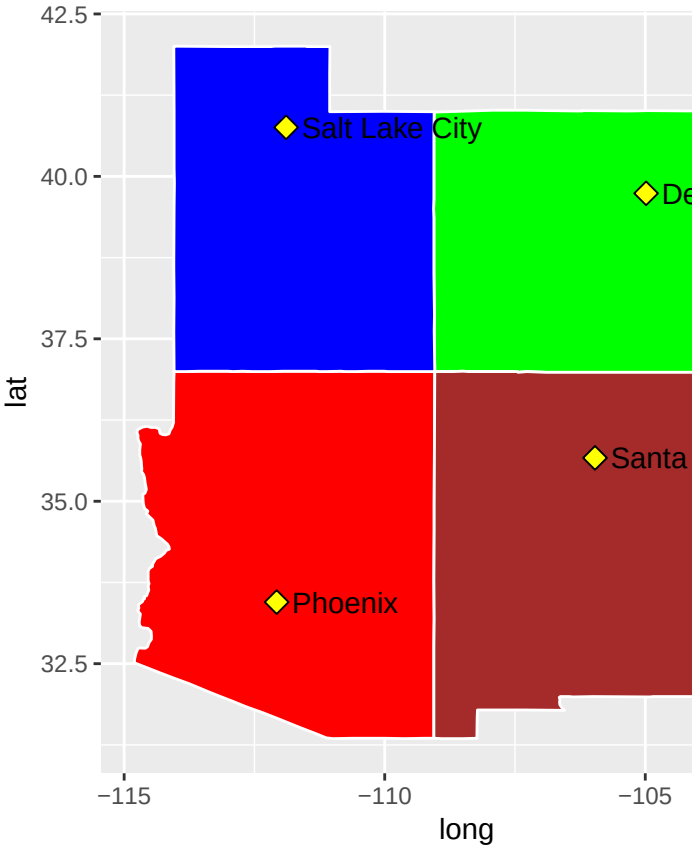
```
## Parsed with column specification:
## cols(
##   state = col_character(),
##   capital = col_character(),
##   lat = col_double(),
##   long = col_double()
## )
```

It looks like this:

```
caps
```

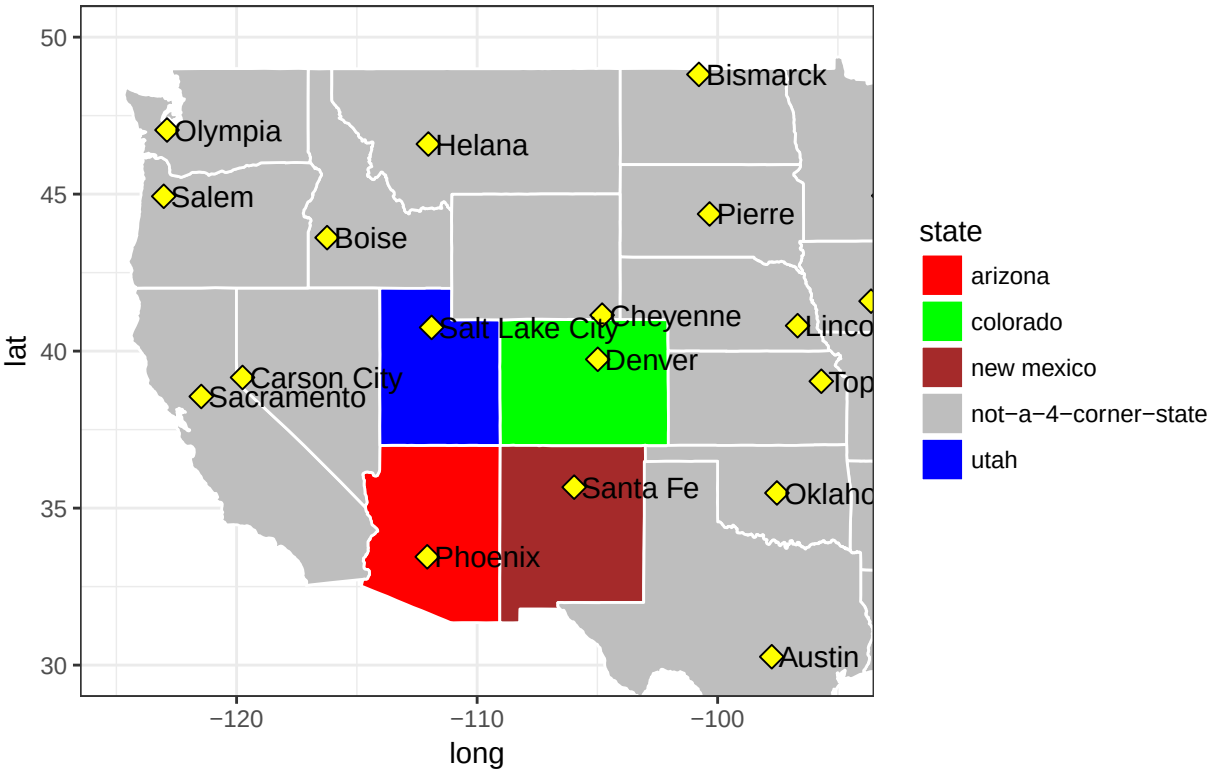
```
## # A tibble: 50 × 4
##          state     capital      lat       long
##          <chr>       <chr>    <dbl>      <dbl>
## 1      Alabama  Montgomery 32.36154  -86.27912
## 2       Alaska      Juneau 58.30194 -134.41974
## 3      Arizona     Phoenix 33.44846 -112.07384
## 4     Arkansas Little Rock 34.73601  -92.33112
## 5   California  Sacramento 38.55560 -121.46893
## 6     Colorado      Denver 39.73917 -104.98417
## 7  Connecticut    Hartford 41.76700  -72.67700
## 8     Delaware       Dover 39.16192  -75.52675
## 9      Florida Tallahassee 30.45180  -84.27277
## 10     Georgia     Atlanta 33.76000  -84.39000
## # ... with 40 more rows
```

You can quickly download the CSV file from here. But, if you downloaded the `inputs` directory and placed it in the right spot to run through the code above, you then already have this.



Once you have that, try to put together a map that looks like this:

OK, now, to display your mad dplyr skills and mapping capabilities, do the gyrations necessary to do this:



8.4.2 Short assignment of plotting with ggmap()

Here are the locations of 357 Wilson's warblers sampled in North America. You can download them from here if you don't already have the `inputs`.

```
wiwa <- readRDS("inputs/breeding_wiwa_isotopes.rds")

# look at the range of the lat longs
range(wiwa$lat)
```

```
## [1] 35.195 63.716
```

```
range(wiwa$long)
```

```
## [1] -157.283  -66.700
```

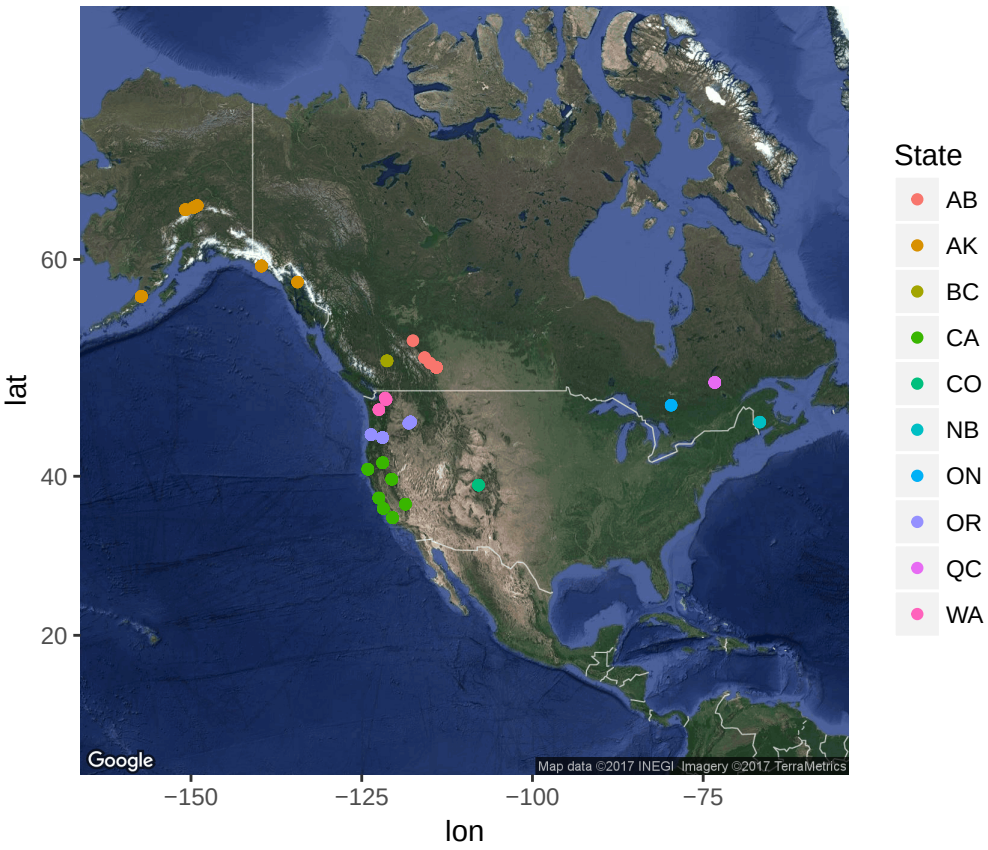
The data frame looks like this:

```
wiwa
```

```
## # A tibble: 357 × 15
##         ID Short_Name Isotope.Value        Source Species Region  Location
##      <chr>      <chr>         <dbl>         <chr>   <chr>  <lgl>     <chr>
## 1  01N0729    wCAHU01     -94.99667 Kelly_Dec2013    WIWA     NA      Hume
## 2  01N0730    wCAHU02    -119.67951 Kelly_Dec2013    WIWA     NA      Hume
## 3  01N0731    wCAHU03    -102.67823 Kelly_Dec2013    WIWA     NA      Hume
## 4  01N0732    wCAHU04    -100.38197 Kelly_Dec2013    WIWA     NA      Hume
## 5  01N0733    wCAHU05    -102.88861 Kelly_Dec2013    WIWA     NA      Hume
## 6  01N7414    wAKUG11    -102.90747 Kelly_Dec2013    WIWA     NA Ugashik_2
## 7  01N7415    wAKUG12    -120.25431 Kelly_Dec2013    WIWA     NA Ugashik_2
## 8  01N7416    wAKUG13    -116.58946 Kelly_Dec2013    WIWA     NA Ugashik_2
## 9  01N7417    wAKUG14    -124.93382 Kelly_Dec2013    WIWA     NA Ugashik_2
## 10 01N7418    wAKUG15    -114.09978 Kelly_Dec2013    WIWA     NA Ugashik_2
## # ... with 347 more rows, and 8 more variables: Near_Town <chr>,
## #   State <chr>, Country <chr>, Kristina_Notes <lgl>, Age <chr>,
## #   lat <dbl>, long <dbl>, stage <chr>
```

Let's make a google_map with some of that info on it—namely the locations of the samples colored by the State or Province that they are in.

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=49,-110&zoom=3&size=640x640&scale=2&ma
```



That shows one of the limitations of plotting things in unprojected latitude and longitude coordinates: Alaska gets all inflated. Next week, if everyone really wants to see how we might overcome that using the Natural Earth Data rasters, we might just be able to do so.

Chapter 9

Plotting "Spatial" Data with ggplot

Note that if you, the student, wish to run all the code yourself, you should download the `inputs` directory as a zipped file by going here with a web browser and then clicking the big "Download" button on the right. Once you have downloaded that, unzip it and put the whole `inputs` directory in the current working directory where you are working with R.

Today we are going to take a very cursory look at R's facilities for handling *spatial data* and ways that it can be plotted using ggplot. When we say "spatial data" we mean data that are associated with a geographical location on the earth, and typically tied to a specific coordinate system that describes their location. Why is this important? It turns out that almost all of the geographical information that you can come by on the Internet is of this type, and furthermore, it is usually stored in a fairly specialized form that will look like Greek to the newcomer. Fortunately, R has a few specialized packages that make it easy to interact with this sort of data, to read it in and write it out, and perform spatial operations.

For years, these types of tasks were typically carried out with proprietary software like Esri's ArcGIS package, where, if you go to their website they will tell you that "making and sharing beautiful maps" is "possible *only* with ArcGIS online." However, you *can* make and share beautiful maps from similar data sets using R, which carries the advantage of being reproducible, open-source, free (as in speech), and free (as in beer). It also, is not too terribly hard to learn to deal with spatial data in R, and it seems to be getting easier all the time. So, if you already are using R for data analysis, the R route for spatial data certainly seems to be a better approach than paying Esri 500 clams a year for an ArcGIS subscription.

In this session we will cover only simple topics (keep in mind that your instructor is a statistical geneticist that only plays with making maps in R on the side), and we will focus almost exclusively on plotting data rather than on spatial operations (like testing if points are in polygons, or clipping, etc). The topics will be:

1. Vector data types
2. Coordinate reference systems
3. Plotting vector data types using `ggplot`
4. Raster data types

You will need a few packages to work with these data. If you don't already have them, you will want to get a few from CRAN:

```
install.packages(c("sp", "rgdal", "raster"))
```

and, though `ggspatial` is available on CRAN, I recommend you get the development version from GitHub:

```
devtools::install_github("paleolimbot/ggspatial")
```

When learning the functions in this spatial-data ecosystem, it is often difficult to keep straight which functions come from which packages, so I will often use namespace addressing (i.e., I will write `rgdal::readOGR` instead

of just `readOGR`, or `ggspatial::geom_spatial` instead of simply `geom_spatial`, etc.) to make that explicit here.

As always, let's load the tidyverse

```
library(tidyverse)
library(rgdal)
library(sp)
library(ggspatial)
```

9.1 Point and Vector Data Types

Geospatial *point* data is fairly straightforward—each datum can be represented as a point on the earth. There are two other main types of what are called "vector" geospatial data. These are *spatial lines* and *spatial polygons*. Spatial lines are just a series of points which, when connected together, form a line. Spatial polygons are just a series of points which, when connected together (and with the last point getting connected back to the first one) subtend an area. In addition to being tied to locations on the earth, each point, or line, or polygon is typically also associated with some other data. For example, a polygon could be the boundary of a county (like we saw in the previous lecture) in which case it could be associated with a name, an area, and a population size, etc. A line might represent a section of a stream, in which case it could be associated with the name of the river, the number of fish per mile, and its average flow, etc.

Data of this sort are often stored in what are called *shapefiles*. As Wikipedia states it, "The shapefile format is a popular geospatial vector data format for geographic information system (GIS) software. It is developed and regulated by Esri as a (mostly) open specification for data interoperability among Esri and other GIS software products. The shapefile format can spatially describe vector features: points, lines, and polygons, representing, for example, water wells, rivers, and lakes."

Data of this type can be read into R using `rgdal::readOGR`. Once it has been read into R, point, line, or polygon data are stored as objects of class: `SpatialPointsDataFrame`, `SpatialLinesDataFrame` or `SpatialPolygonDataFrame`, respectively.

9.1.1 Some example vector data

I have downloaded several small data sets about Santa Cruz that are available, free and open, from the county GIS website.

I have them in the `inputs` directory of the course webpage repository. Follow the instructions at the top of this lecture to get them all. The data sets that we will play with are:

- Street_Lights — spatial points giving the location of street lights in the county
- Streams — spatial lines showing rivers and tributaries
- Watersheds — spatial polygons showing watershed boundaries

Shapefiles are actually directories that contain a number of different files with different extensions inside them. Let's look at the Watersheds shapefile for an example:

```
dir("inputs/santa-cruz-county/Watersheds/")
```

```
## [1] "Watersheds.cpg" "Watersheds.dbf" "Watersheds.prj" "Watersheds.shp"
## [5] "Watersheds.shx"
```

These different files carry all the information. Some of them are text readable, like the `.prj` file, but many of the others are compressed binary files. Fortunately, we don't ever have to open them up and try to read them with a text editor, as we can just read them in with `rgdal::readORG`.

9.1.2 Reading in Vector Spatial Data

The `readOGR` function in `rgdal` does this for us. It takes as the first argument (the `dsn` parameter) the *directory* that holds the shapefile. In the above case that would be `"inputs/santa-cruz-county/Watersheds"`. The second argument (the `layer` parameter) is the prefix of the name of the files that you want to read. Above, all the files started with `Watersheds` so this second argument is `Watersheds`. Let's do it for the street lights, streams, and watersheds:

```
lights <- rgdal::readOGR(dsn = "inputs/santa-cruz-county/Street_Lights", layer = "Street_Lights")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "inputs/santa-cruz-county/Street_Lights", layer: "Street_Lights"
## with 2959 features
## It has 36 fields
## Integer64 fields read as strings:  OBJECTID OP_SCHED
```

```
streams <- rgdal::readOGR(dsn = "inputs/santa-cruz-county/Streams", layer = "Streams")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "inputs/santa-cruz-county/Streams", layer: "Streams"
## with 1514 features
## It has 8 fields
## Integer64 fields read as strings:  OBJECTID
```

```
watersheds <- rgdal::readOGR(dsn = "inputs/santa-cruz-county/Watersheds", layer = "Watersheds")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "inputs/santa-cruz-county/Watersheds", layer: "Watersheds"
## with 17 features
## It has 6 fields
## Integer64 fields read as strings:  OBJECTID COUNT_
```

When the data are read in, a message about what is in the data set is printed.

Note that the files inside the directory will not always have the same name as the directory. For example, inside the "Salmon_Streams" shapefile directory you have:

```
dir("inputs/santa-cruz-county/Salmon_Streams")
```

```
## [1] "Fishery_Resource.cpg" "Fishery_Resource.dbf" "Fishery_Resource.prj"
## [4] "Fishery_Resource.shp" "Fishery_Resource.shx"
```

Exercise: read the Salmon_Streams data into a variable called `salmon_streams`.

A side-note here: `readOGR` does not seem to do proper tilde expansion of file names, so trying to read something on your hard drive that you might usually do, with a tilde giving your home directory, will fail:

```
# this fails
stream_fail <- rgdal::readOGR(dsn = "~/Documents/git-repos/rep-res-eeb-2017/inputs/santa-cruz-county/St
                              layer = "Streams")
```

If you want to do that, you need to wrap the directory path inside the `normalizePath()` function, like this:

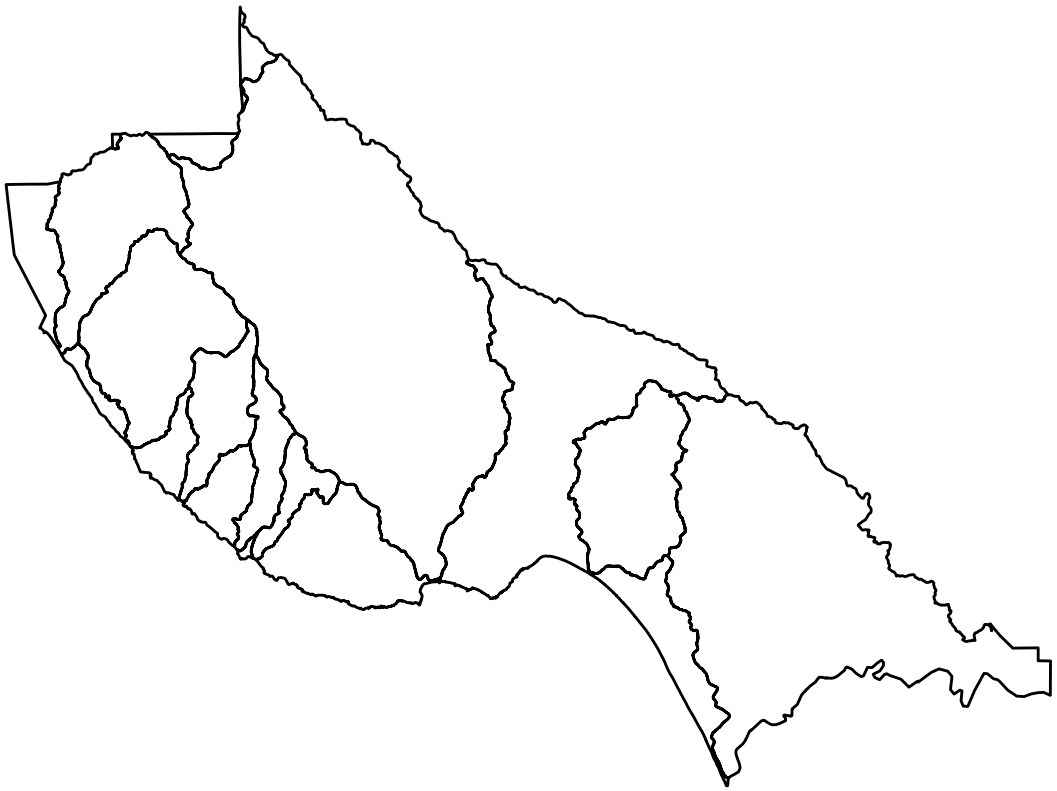
```
# this doesn't fail on my laptop
stream_no_fail <- rgdal::readOGR(dsn = normalizePath("~/Documents/git-repos/rep-res-eeb-2017/inputs/san
                              layer = "Streams")
```

9.1.3 Quick Visualization of our Vector Data

Before we start drilling down into these data sets to see how `Spatial{Points,Lines,Polygons}DataFrame`s are structured, let's just plot them quickly. We will use the `ggspatial` package which extends `ggplot2` to spatial data with a nice syntax. Basically, that package provides a new geom called `geom_spatial()` that deals with plotting spatial points, lines, or polygons. The same function `geom_spatial` is used to plot points, lines, or polygons, It "looks at"" the spatial data set it is provided to determine whether it should plot points, lines, or polygons.

Here are the watershed boundaries:

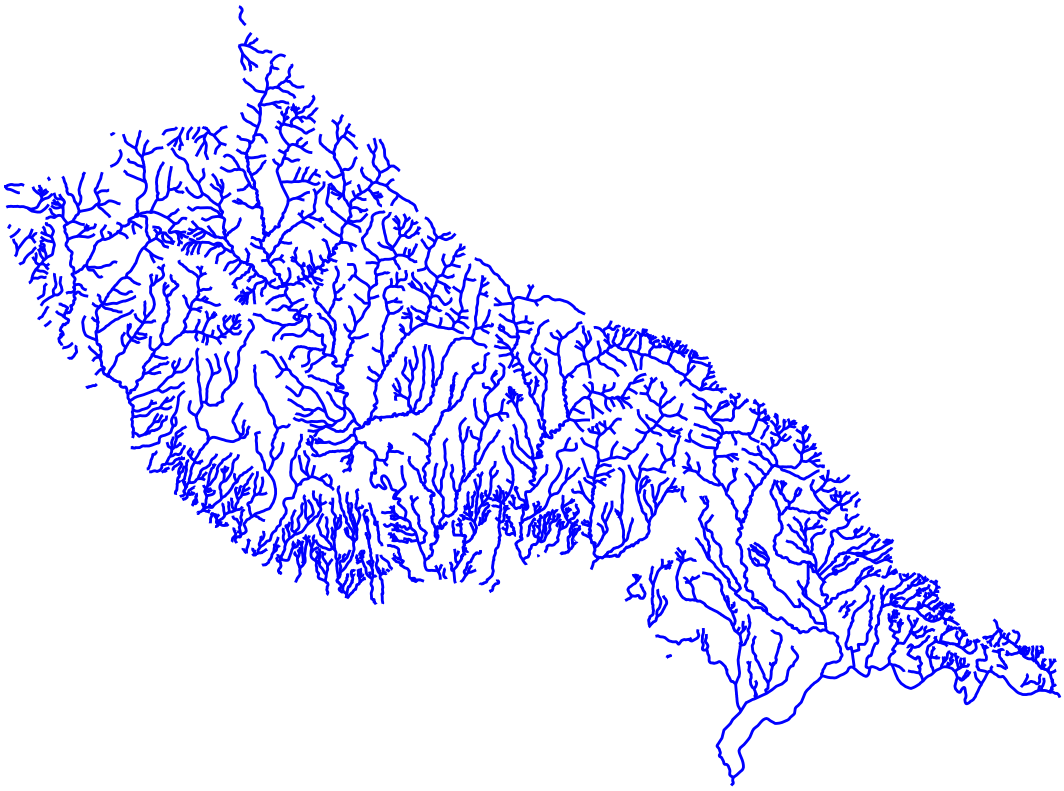
```
ggplot() +
  ggspatial::geom_spatial(data = watersheds, fill = NA, colour = "black") +
  theme_void() +
  coord_map()
```



OK, that is pretty cool. See that these are, as we expect, polygons.

Let's do the streams:

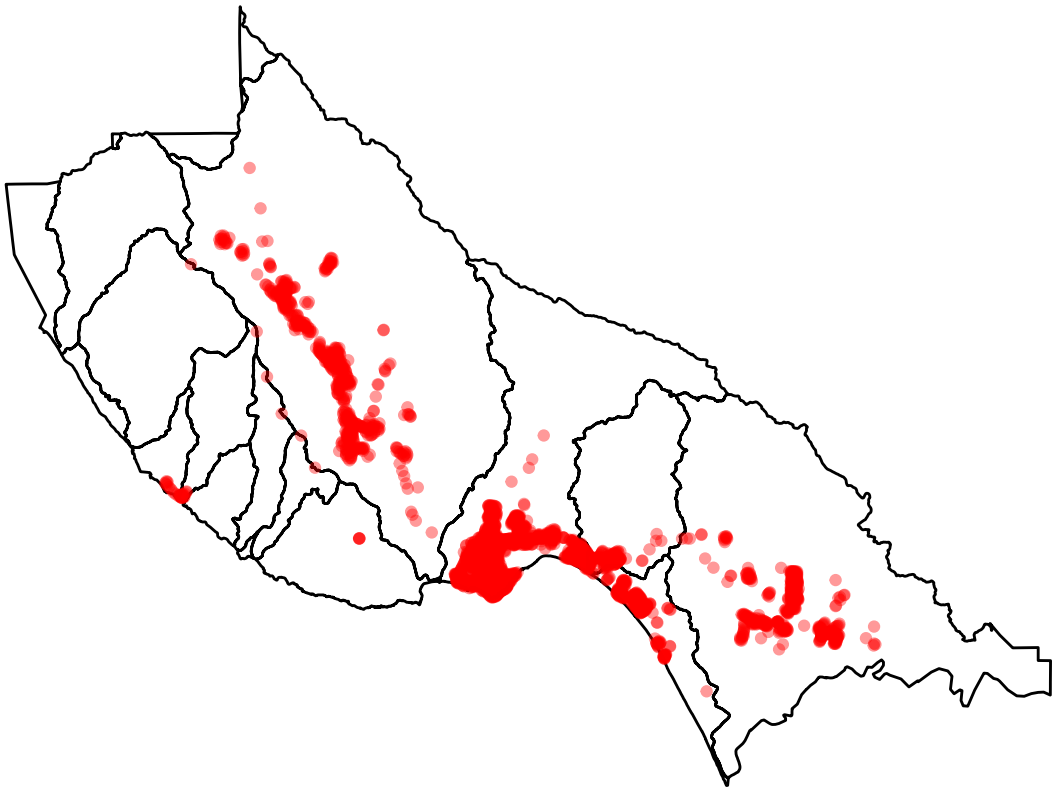
```
ggplot() +
  ggspatial::geom_spatial(data = streams, colour = "blue") +
  theme_void() +
  coord_map()
```



Yep, those are lines!

Finally, let's plot the street lights, but we will put them over the top of the watersheds so that we have a little context for them.

```
ggplot() +
  ggspatial::geom_spatial(data = watersheds, fill = NA, colour = "black") +
  ggspatial::geom_spatial(data = lights, colour = "red", alpha = 0.4) +
  theme_void() +
  coord_map()
```



OK, those are points. That makes sense.

9.1.4 The Structure of Spatial*DataFrames

When we write `Spatial*DataFrames` we mean, collecively, `SpatialPointsDataFrames`, `SpatialLinesDataFrames`, and `SpatialPolygonsDataFrames`. These are the types of objects that R uses to store spatial information. Each of these is what is called an S4 object type. S4 objects are like lists in R, but instead of having elements you access with `$`, you have information in "slots" that you access with `@`. (What?! OK, don't worry too much, you won't typically ever have to access these slots using `@`, but I do want to use it to talk about the underlying structure of these beasts.)

First, let's look at the slot names in these three different types of Spatial Data Frames:

```
# spatial points
slotNames(lights)
```

```
## [1] "data"        "coords.nrs"  "coords"      "bbox"        "proj4string"
```

```
# spatial lines
slotNames(streams)
```

```
## [1] "data"        "lines"       "bbox"        "proj4string"
```

```
# spatial polygons
slotNames(watersheds)
```

```
## [1] "data"        "polygons"    "plotOrder"   "bbox"        "proj4string"
```

Aha! They all have:

- `data` which is a data frame of information about each feature
- `bbox` which is information about the spatial extent of features in the file
- `proj4string` which holds information about the projection (coordinate reference system)

Then, the spatial points have `coords` while the spatial lines have `lines` and the spatial polygons have `polygons`.

Let us look more closely at the `data` slot in `streams`. This is just a data frame, and so we can look at it as a tibble:

```
as_tibble(streams@data)
```

```
## # A tibble: 1,514 × 8
##    OBJECTID       STREAM_NM   STREAMTYPE  GNIS_ID     ID ENABLED  Named
## *    <fctr>          <fctr>       <fctr>   <fctr> <fctr>  <fctr> <fctr>
## 1         1     Adams Creek INTERMITTENT       NA      0      NA    Yes
## 2         2      Alba Creek    PERENNIAL 00218063      1      NA    Yes
## 3         3     Amaya Creek    PERENNIAL 00218217      2      NA    Yes
## 4         4 Ano Nuevo Creek    PERENNIAL 00254567      3      NA    Yes
## 5         5     Aptos Creek    PERENNIAL 00254571      4      NA    Yes
## 6         6 Archibald Creek INTERMITTENT 00218350      5      NA    Yes
## 7         7   Baldwin Creek    PERENNIAL 00218614      6      NA    Yes
## 8         8     Bates Creek    PERENNIAL 00218726      7      NA    Yes
## 9         9      Bean Creek    PERENNIAL 00218782      8      NA    Yes
## 10       10   Bennett Creek    PERENNIAL 00219040      9      NA    Yes
## # ... with 1,504 more rows, and 1 more variables: SHAPESTLen <dbl>
```

And what about the `lines` slot? Well, that is a spatial data object that holds the coordinates that you would connect to make each line. This is a list of things, in which each element corresponds to a row in `data`.

9.1.5 Subsetting Spatial*DataFrames

If you want to pick out only some of the features in a Spatial*DataFrame, based on the values in the `data`, you can use `subset`. It works a little like `filter` from the tidyverse.

For an example, let's see if we can keep only those stream segments that are classified as INTERMITTENT:

```
intermitt <- streams %>%
  subset(., STREAMTYPE == "INTERMITTENT")
```

While there were 1514 features in `streams`

```
length(streams)
```

```
## [1] 1514
```

In `intermitt` there are only 290:

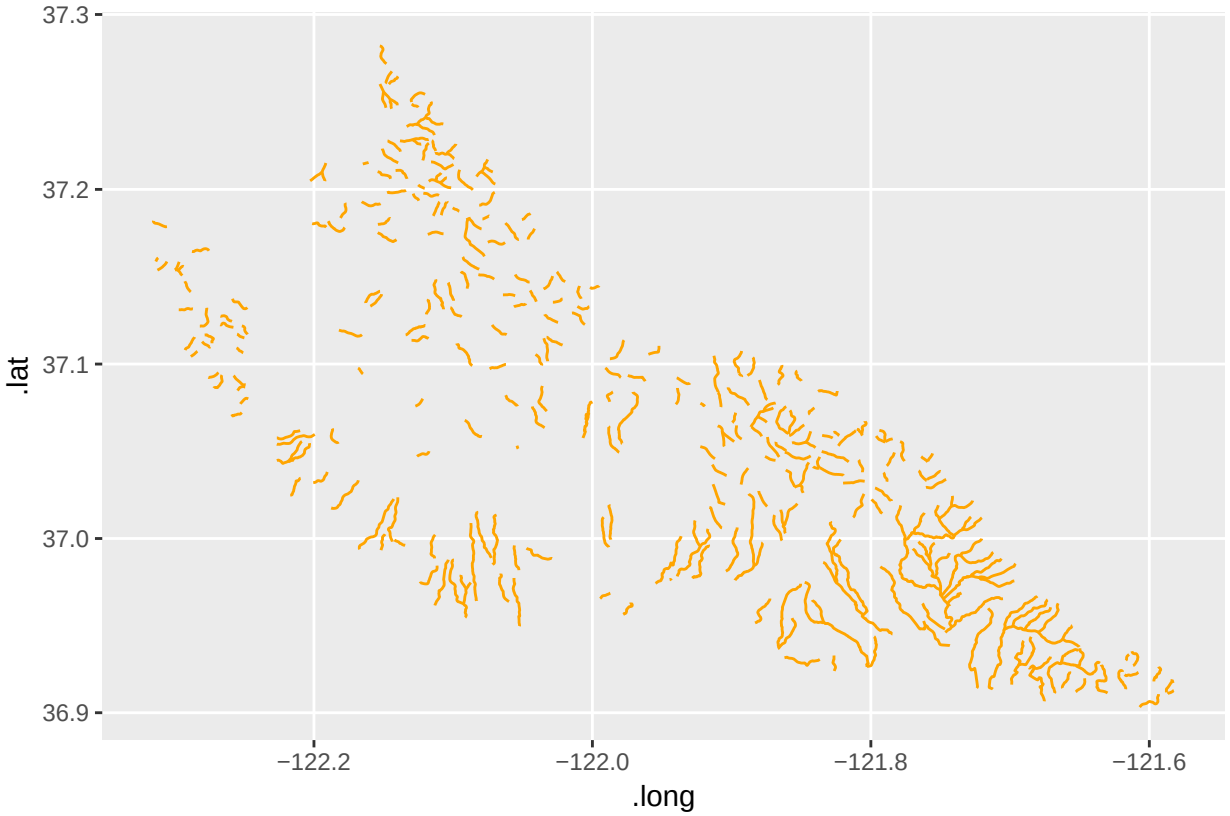
```
length(intermitt)
```

```
## [1] 290
```

We can plot those to see if the spatial distribution of intermittent streams makes sense to us:

```
ggplot() +
  ggspatial::geom_spatial(data = intermitt, colour = "orange") +
  coord_map()
```

```
## Converting coordinates to lat/lon
```



Yep, those are creeks that are mostly up higher in the watersheds.

9.2 Coordinate Reference Systems

I am not going to say too much about this. Anna Nisi directed me to a fantastic three-page primer on CRSs in R, and I direct people there.

`ggspatial` by default converts everything to lat/lon coordinates, so you can almost get away with ignoring it, but you should read about it.

To learn about the CRS of a Spatial*DataFrame you can do like this:

```
proj4string(streams)
```

```
## [1] "+proj=lcc +lat_1=37.06666666666667 +lat_2=38.43333333333333 +lat_0=36.5 +lon_0=-120.5 +x_0=2000000
```

That shows us it is a Lambert Conformal Conic projection ("lcc").

If we wanted to transform the coordinate reference system to something else, we can do like this:

```
streams_lat_lon <- spTransform(streams, CRS("+init=epsg:4326"))
```

Check it:

```
proj4string(streams_lat_lon)
```

```
## [1] "+init=epsg:4326 +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

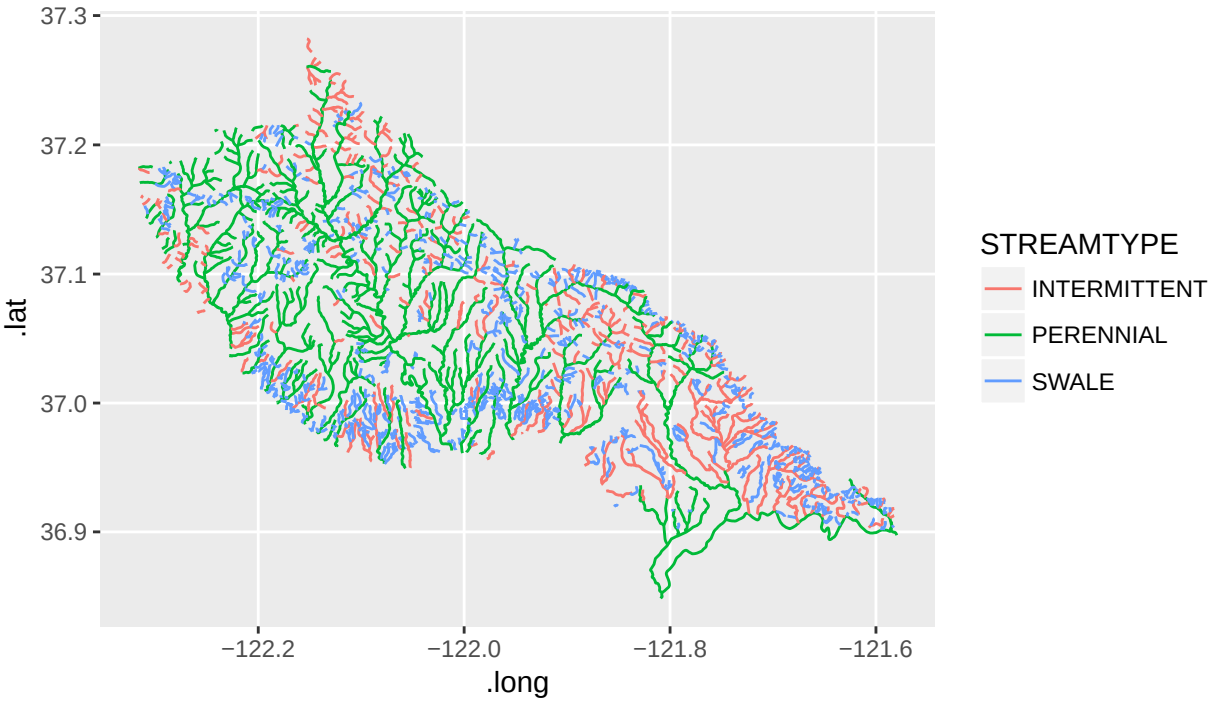
Yep, that is right.

9.3 Plotting things with ggspatial

For a good primer on `ggspatial` find the README at its GitHub Page

The ggspatial package lets you plot spatial objects using a gpplot-like syntax. You can map aesthetics to the columns in the `data` slot. For example, let's plot `streams` and color them by STREAMTYPE:

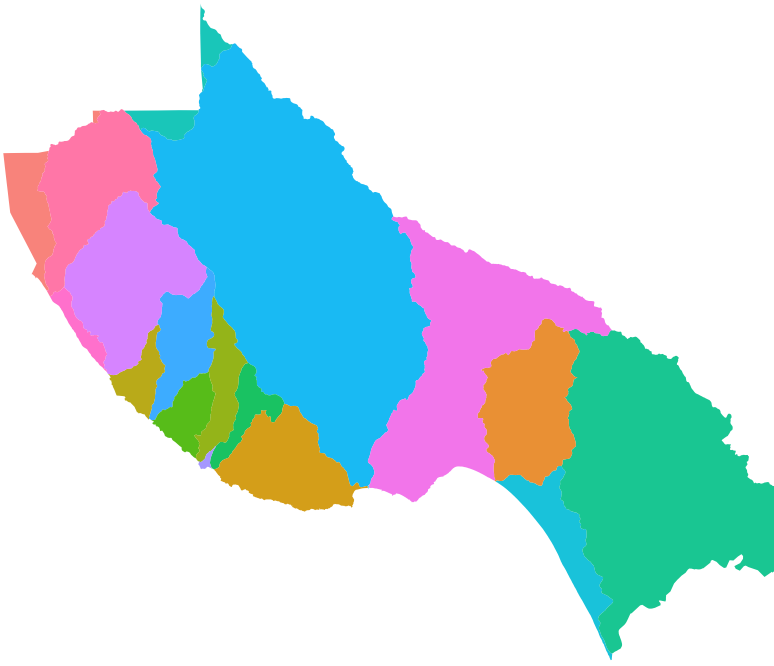
```
ggplot() +
  ggspatial::geom_spatial(data = streams, mapping = aes(colour = STREAMTYPE)) +
  coord_map()
```



That's cool.

Exercises

1. Look at the `data` field of `watersheds`



2. Then plot the watersheds in different colors like this:

9.3.1 Want a background?

How about an open streets background layer? You can add it with a `ggspatial::geom_osm()` layer. Different types of maps are available:

```
# different types of OSM maps available:
rosm::osm.types()
```

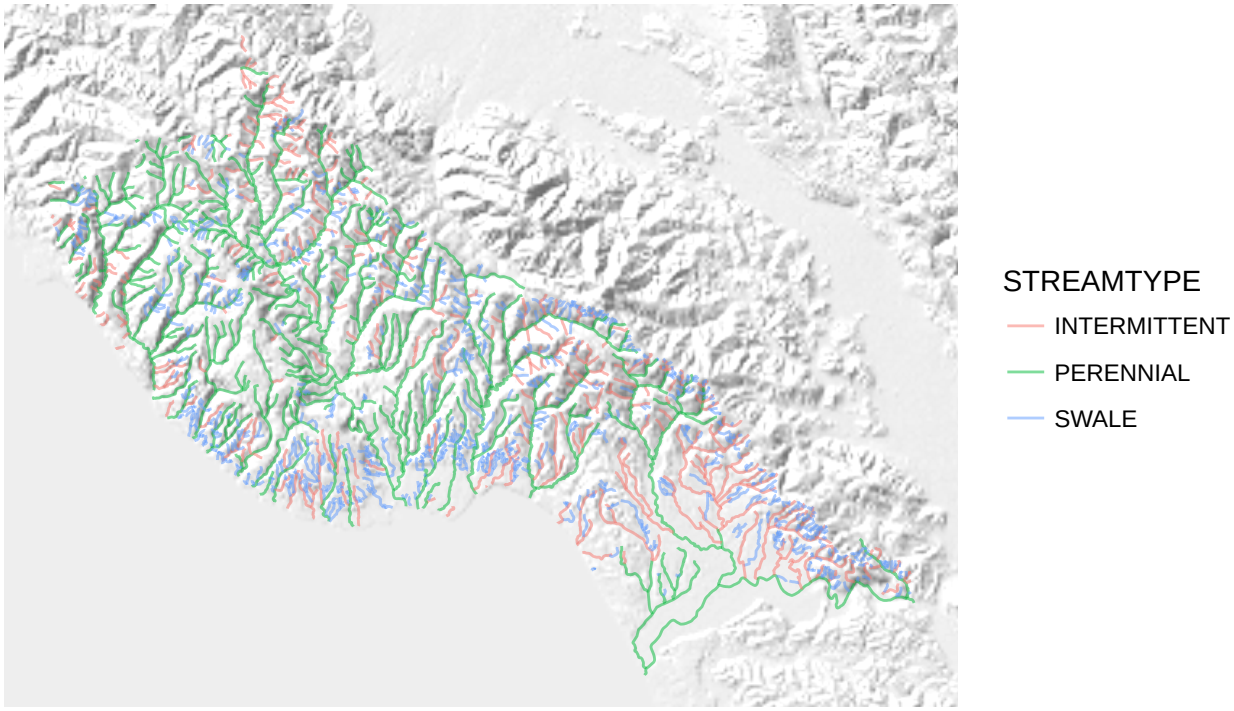
```
##  [1] "osm"                    "opencycle"
##  [3] "hotstyle"               "loviniahike"
##  [5] "loviniacycle"           "hikebike"
##  [7] "hillshade"              "osmgrayscale"
##  [9] "stamenbw"               "stamenwatercolor"
## [11] "osmtransport"           "thunderforestlandscape"
## [13] "thunderforestoutdoors"  "cartodark"
## [15] "cartolight"
```

So, let's put a hillshade background on:

```
ggplot() +
  geom_osm(type = "hillshade") +
  ggspatial::geom_spatial(data = streams, mapping = aes(colour = STREAMTYPE), alpha = 0.5) +
  coord_map() +
  theme_void()
```

```
## Converting coordinates to lat/lon
```

```
## Zoom: 10
```



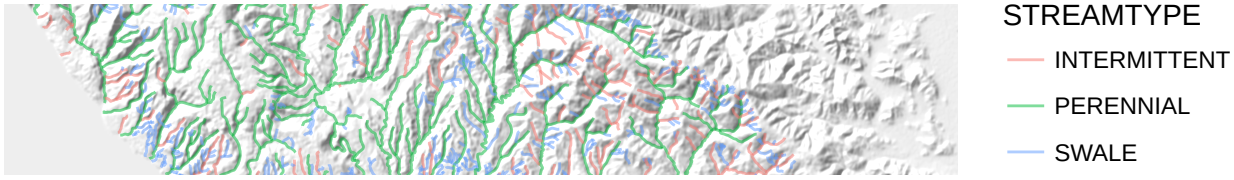
Cowabunga!

Using ggspatial with open street maps is a bit nicer than using `ggmap` because we are not bound to having square plots and it looks like it figures out the appropriate zoom for the backgound, etc. Behold the thin slice of shaded hillside background!

```
ggplot() +
  geom_osm(type = "hillshade") +
  ggspatial::geom_spatial(data = streams, mapping = aes(colour = STREAMTYPE), alpha = 0.5) +
  coord_map(xlim = c(-122.3, -121.6), ylim = c(37.0, 37.1)) +
  theme_void()
```

```
## Converting coordinates to lat/lon
```

```
## Zoom: 12
```



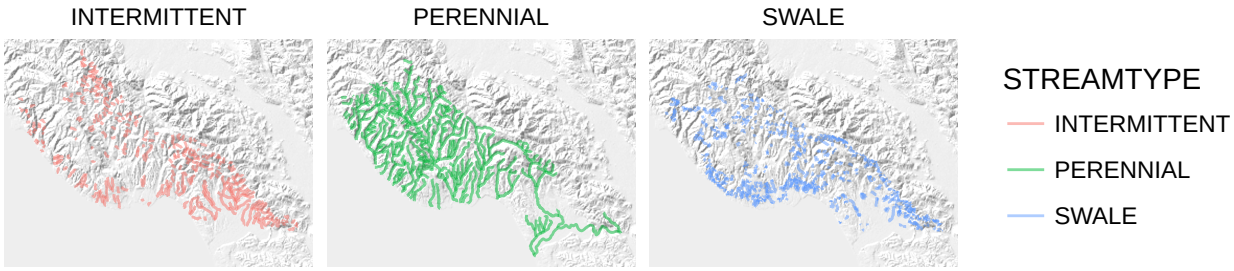
It is worth pointing out that you will very often use `coord_map()` when working with `ggspatial`, especially when using Open Street Maps layers under them.

Check this out, even faceting seems to work with ggspatial.

```
ggplot() +
  geom_osm(type = "hillshade") +
  ggspatial::geom_spatial(data = streams, mapping = aes(colour = STREAMTYPE), alpha = 0.5) +
  coord_map() +
  theme_void() +
  facet_wrap(~ STREAMTYPE)
```

```
## Converting coordinates to lat/lon
```

```
## Zoom: 10
## Zoom: 10
## Zoom: 10
```



These background layers are received as a type of spatial data called a *spatial raster*. You can think of rasters as digital "images"—basically regular grids of pixels.
`ggspatial` has exploited one of ggplot's facilities for efficiently plotting images to be able to efficiently plot rasters in the background. It also provides a function, `geom_spraster()` that let's us do the same with spatial rasters from anywhere. This is a great feature that also let's us plot rasters that have been projected to different coordinate reference systems. Before we get into this, let's learn a little about rasters.

9.4 Spatial Raster Data

R has a fantastic package, called `raster`, written by Robert Hijmans (who was a collaborator with Kristen when they were both at Berkeley, check this out!). The `raster` package provides a nice interface for dealing with spatial raster types and doing a variety of operations with them.

We are going to start with an example: shaded relief of Carmel Bay avaiable through NOAA's Digital Elevation Model Global Mosaic (Color Shaded Relief). I have already downloaded it to the `inputs` directory because, to be quite honest, obtaining it through R was not as straightforward as I would have hoped. This raster has multiple layers. It is a color image stored as a multi-layer (or "band") file. Accordingly we can use the `brick` function to read it in as a "rasterBrick":

```
carmel_bay <- raster::brick("inputs/carmel_bay_bathy.tif")
```

Once we have done that we can read about it by printing it:

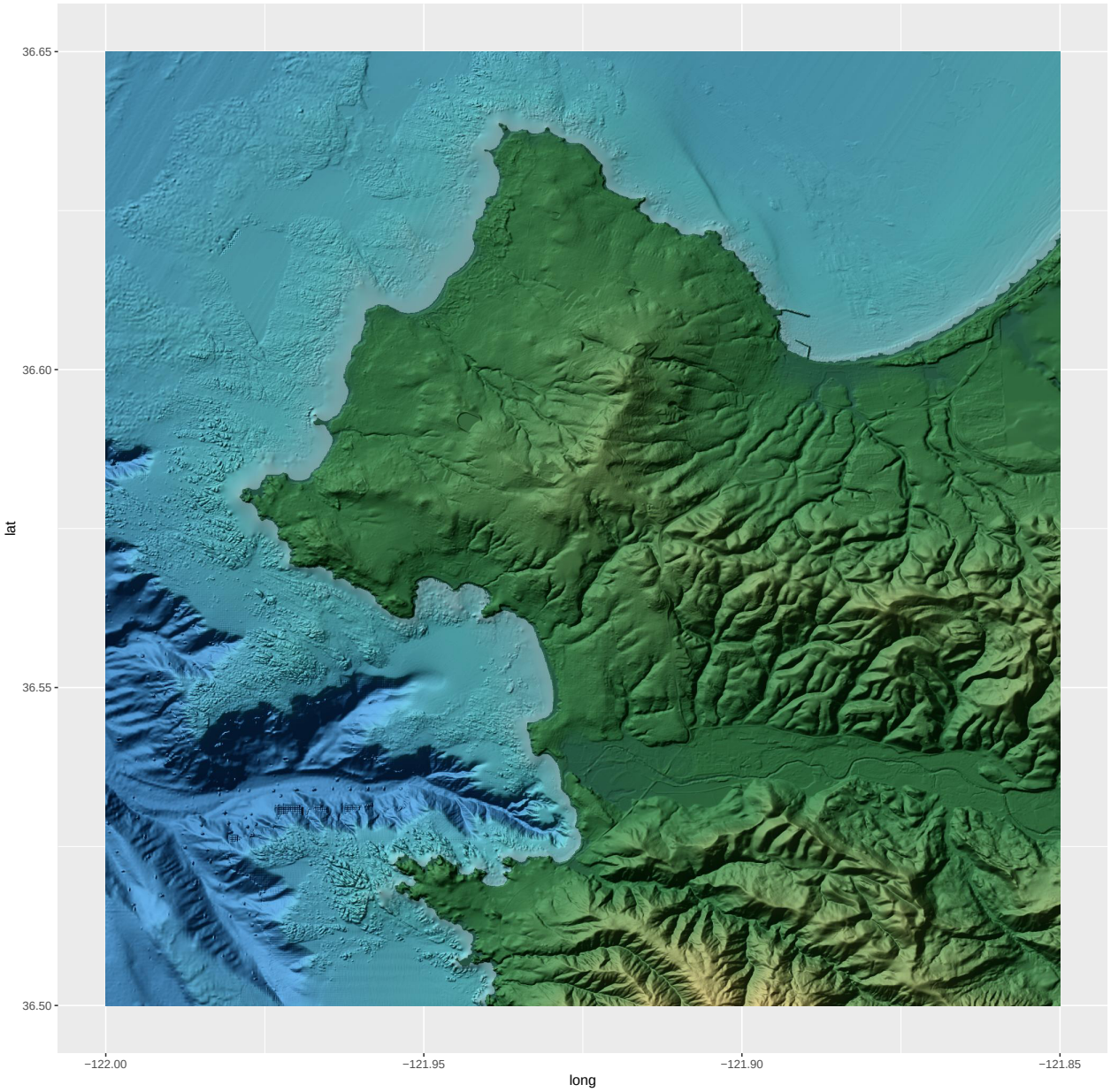
```
carmel_bay
```

```
## class       : RasterBrick
## dimensions  : 1800, 1800, 3240000, 3  (nrow, ncol, ncell, nlayers)
## resolution  : 8.333333e-05, 8.333333e-05  (x, y)
## extent      : -122, -121.85, 36.5, 36.65  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## data source : /Users/eriq/Documents/git-repos/rep-res-eeb-2017/inputs/carmel_bay_bathy.tif
## names       : carmel_bay_bathy.1, carmel_bay_bathy.2, carmel_bay_bathy.3
## min values  :                  0,                  4,                  0
## max values  :                230,                242,                253
```

That tells us a lot of useful things, like (from the "dimensions" line) there are 3 layers, each with 3.24 million cells, on a grid that is 1800 x 1800 cells. It also gives us information about the coordinate reference system (on the "coord. ref." line).

That is all well and good. Now, let us see what that looks like. `ggspatial` has the function `geom_spraster_rgb()` for plotting the entire extent of a three-banded raster, interpreting the bands as red, green and blue.

```
ggplot() +
  ggspatial::geom_spraster_rgb(carmel_bay) +
  coord_fixed()
```



That is pretty, and could conceivably make a nice background for some of Diana's rockfish plots.

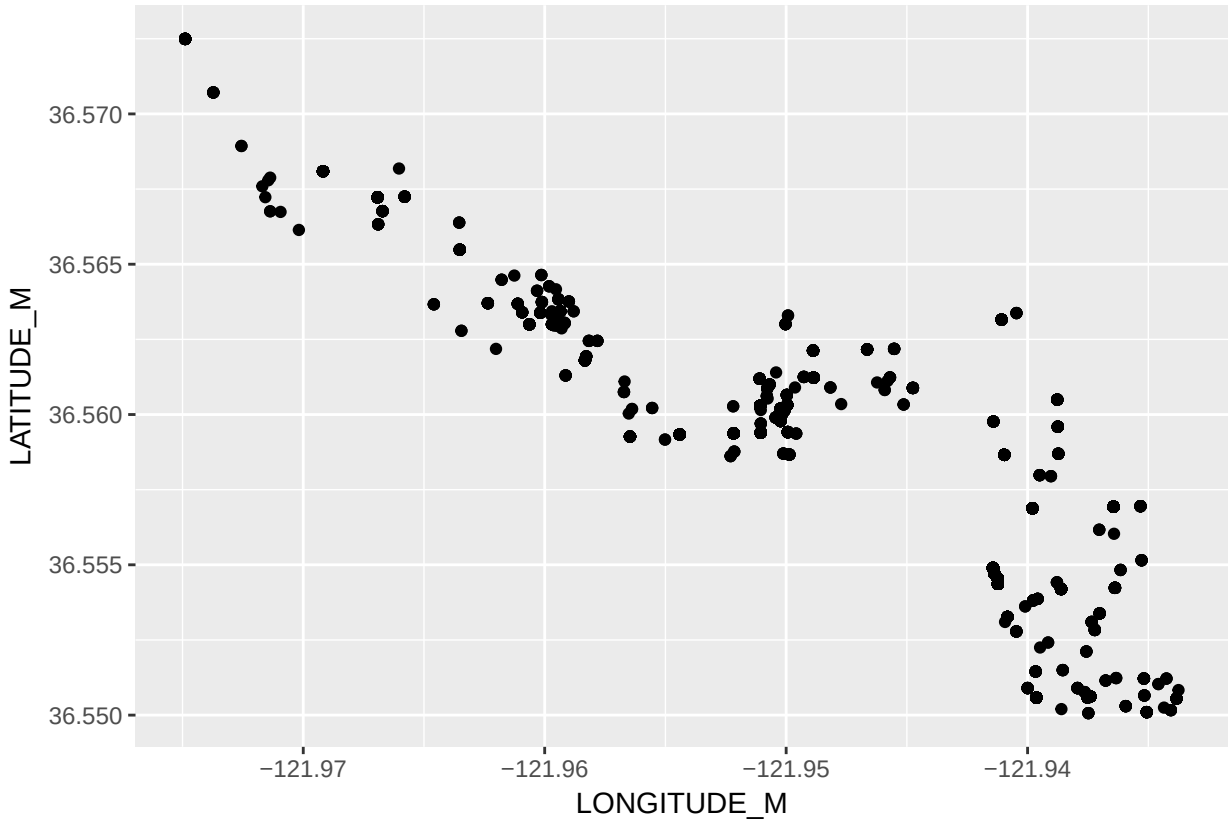
There is another function in `ggspatial` called `annotation_spraster` that plots a raster, but does not change the plot boundaries. This is very useful if you have a lot of points that you wish to plot, and you want the plot boundaries to be sized to contain all your points, and you, accordingly, only want that particular piece of your background raster in the plot. Let's see it in action by grabbing Diana's rockfish data, but filtering it only to those points in the Stillwater Cove area.

```
sebastes_stillwater <- readRDS("inputs/sebastes_locations.rds") %>%
  filter(LATITUDE_M > 36.55,
         LATITUDE_M < 36.575,
         LONGITUDE_M > -121.975,
```

```
          LONGITUDE_M < -121.925)
```

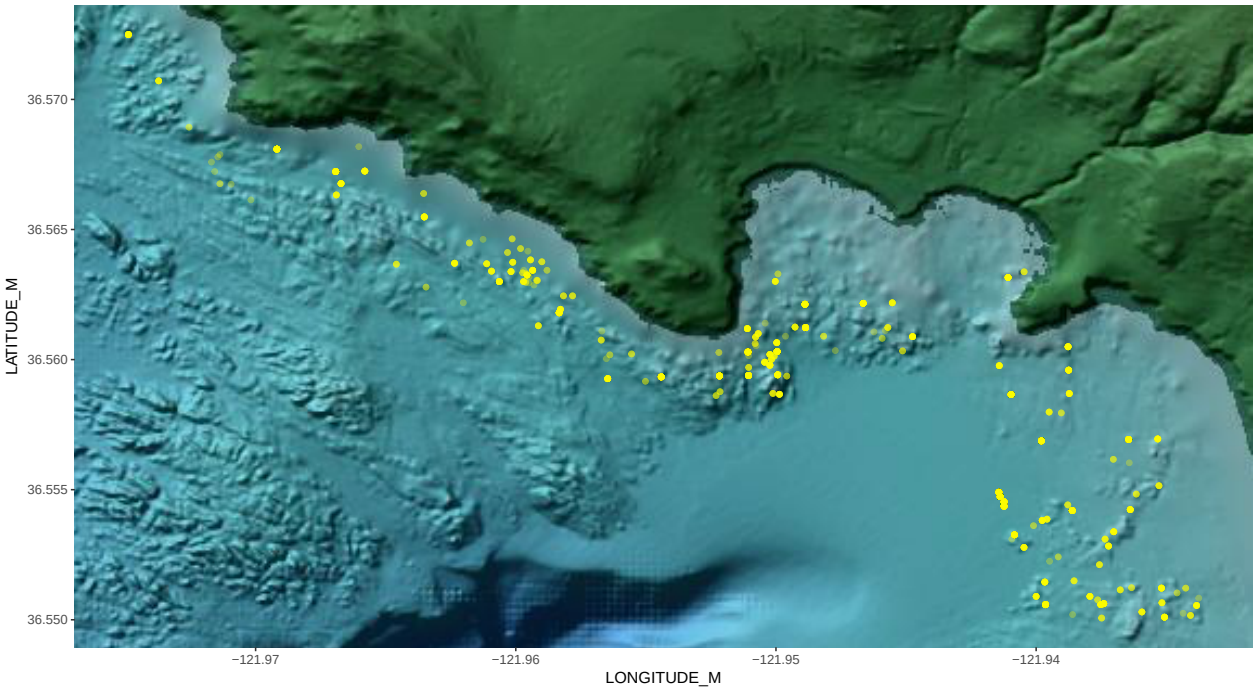
Then plot those. Here they are by themselves:

```
ggplot() +
  geom_point(data = sebastes_stillwater, mapping = aes(x = LONGITUDE_M, y = LATITUDE_M)) +
  coord_quickmap()
```



And here they are with the raster in the background:

```
ggplot() +
  ggspatial::annotation_spraster(carmel_bay, interpolate = TRUE) +
  geom_point(data = sebastes_stillwater, mapping = aes(x = LONGITUDE_M, y = LATITUDE_M),
             colour = "yellow",
             alpha = 0.3) +
  coord_fixed()
```



That is pretty cool. It might have been nice to have downloaded a higher resolution raster, which is available, but would have been quite large at the full, zoomed out scale.

One very important thing to note here is that when you are using `ggspatial` you can still plot regular `ggplot2` geoms on top of it. We happened to have some points in a tibble (not, a SpatialPointsDataFrame) with Latitudes and Longitudes, so we just hucked 'em on there using `geom_point`.

9.5 Wrapping Up

Well, we merely scratched the surface of handling and plotting spatial data in R. There are a lot of resources out there:

- RSpatial Book/Tutorial This is an outstanding contribution from Robert Hijmans (it appears). It looks like it is still in progress, but it is quite complete and lucid.
- A CRAN tutorial on spatial data in R
- Melanie Frazier's kick-ass into to coordinate reference systems
- R Spatial Cheatsheet loaded with good reminders of how to do things
- Another Cheat Sheet Great to read through to quickly get a sense of what is possible.
- ArcGIS_to_R_Spatial_CheatShee another cheat sheet well-suited to people who have used the ArcGIS point-and-click GUI.
- Remote sensing in R reference card Superb summary from the folks who bring you the book Remote Sensing and GIS for Ecologists Using Open Source Software which looks like it would be a superb read if you really wanted to dive deeply into this topic.

On top of that, we haven't even touched on interactive maps with R. Those interested in that should check out Leaflet for R.