Introduction to Open Data Science

The Ocean Health Index Team

2017-11-27

Contents

1 Welcome **5**

2 Overview **7**
- 2.1 What to expect . . . 7
- 2.2 Gapminder data: . . . 9
- 2.3 By the end of the course… . . . 9
- 2.4 Prerequisites . . . 9
- 2.5 Credit . . . 9

3 R & RStudio, Rmarkdown **11**
- 3.1 Overview . . . 11
- 3.2 Why learn R with RStudio . . . 11
- 3.3 R at the console, RStudio goodies . . . 12
- 3.4 R functions, help pages . . . 15
- 3.5 Clearing the environment . . . 17
- 3.6 RMarkdown . . . 18
- 3.7 Troubleshooting . . . 20

4 GitHub **23**
- 4.1 Objectives & Resources . . . 23
- 4.2 Why should scientists use Github? . . . 24
- 4.3 Setup Git & GitHub . . . 26
- 4.4 Create a repository on Github.com . . . 27
- 4.5 Clone your repository using RStudio . . . 29
- 4.6 Inspect your repository . . . 33
- 4.7 Add files to our local repo . . . 33
- 4.8 Sync from RStudio to GitHub . . . 36
- 4.9 Explore remote Github . . . 38
- 4.10 Create a new R Markdown file . . . 39
- 4.11 Committing - how often? Tracking changes in your files . . . 40
- 4.12 Troubleshooting . . . 42

5 Visualizing: `ggplot2` **43**
- 5.1 Objectives & Resources . . . 43
- 5.2 Install our first package: `tidyverse` . . . 43
- 5.3 Plotting with **`ggplot2`** . . . 44
- 5.4 Data . . . 45
- 5.5 Building your plots iteratively . . . 47
- 5.6 Faceting . . . 53
- 5.7 **`ggplot2`** themes . . . 56
- 5.8 Geometric objects (geoms) . . . 56
- 5.9 Customization . . . 62

5.10 Bar charts . . . 64
5.11 Arranging and exporting plots . . . 70
5.12 Save and push to GitHub . . . 70

6 Data Wrangling: `dplyr` 71
6.1 Overview of `dplyr` . . . 71
6.2 Prerequisites . . . 71
6.3 Tidy Data . . . 72
6.4 Explore the gapminder data.frame . . . 74
6.5 `dplyr` basics . . . 76
6.6 `filter()` subsets data row-wise (observations). . . . 76
6.7 Your turn . . . 77
6.8 Meet the new pipe `%>%` operator . . . 77
6.9 `select()` subsets data column-wise (variables) . . . 78
6.10 `mutate()` adds new variables . . . 79
6.11 `group_by()` operates on groups . . . 80
6.12 `arrange()` orders columns . . . 81
6.13 All together now . . . 82
6.14 Key Points . . . 83

7 Data Wrangling: `tidyr` 85
7.1 Overview . . . 85
7.2 `tidyr` basics . . . 85
7.3 Explore gapminder data — wide format. . . . 87
7.4 `gather()` data from wide to long format . . . 87
7.5 `spread()` data from long to intermediate format . . . 90
7.6 Your turn . . . 92
7.7 Other links . . . 93

8 Programming 95
8.1 Objectives and Resources . . . 95
8.2 Naming files . . . 95
8.3 Analysis plan . . . 96
8.4 Create an R script . . . 96
8.5 Automation with for loops . . . 96
8.6 Conditional statements with `if` and `else` . . . 102
8.7 Joining datasets . . . 105
8.8 More R! . . . 108
8.9 Ideas for Extended Analysis 2 . . . 108

9 Collaborate with GitHub 109
9.1 Overview . . . 109
9.2 create gh-pages repo and give someone permission . . . 109
9.3 clone to a new Rproject . . . 109
9.4 Rmd analysis together . . . 109

Chapter 1

Welcome

Welcome. This training program is under active development and testing.

This 2-day training workshop will introduce you to open data science so you can work with data in an open, reproducible, and collaborative way. Open data science means that methods, data, and code are available so that others can access, reuse, and build from it without much fuss. Here you will learn a workflow with R, RStudio, Git, and GitHub, as we describe in Lowndes *et al.* 2017: Our path to better science in less time using open data science tools.

This workshop is going to be fun, because learning these open data science tools and practices is empowering! This training book is written so you can use it as self-paced learning, or it can be used to teach an in-person workshop. Either way, you should do everything hands-on on your own computer as you learn.

Before you begin, be sure you are all set up: see Chapter 3.1: Overview and Prerequisites.

Suggested breakdown for a 2-day workshop:

time	Day 1	Day 2
9-10:30	Motivation, R & RStudio, Rmarkdown	Data Wrangling: `tidyr`
break		
11-12:30	GitHub	Programming
lunch		
13:30-15:00	Visualization: `ggplot2`	Collaborating with GitHub
break		
15:30-17:00	Data Wrangling: `dplyr`)	TBD…

License

Chapter 2

Overview

Welcome.

This is a 2-day training workshop to learn R, RStudio, Git, and GitHub, and it's going to be fun and empowering. You will learn a reproducible workflow that can be used in analyses of all kinds, including Ocean Health Index assessments. This is really powerful, cool stuff, and not just for data: I made and published this book using those four tools and workflow.

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

2.1 What to expect

This is going to be a fun workshop.

The plan is to expose you to a lot of great tools that you can have confidence using in your research. You'll be working hands-on and doing the same things on your own computer as we do live on up on the screen. We're going to go through a lot in these two days and it's less important that you remember it all. More imporatantly, you'll have experience with it and confidence that you can do it. The main thing to take away is that there *are* good ways to approach your analyses; we will teach you to expect that so you can find what you need and use it! And, you can use these materials as a reference as you go forward with your analyses.

We'll be talking about :

- how to THINK about data. And not just any data; tidy data.
- how to increase reproducibility in your science
- how to more easily collaborate with others–including your future self!
- how the #rstats community is fantastic. The tools we're using are developed by real people. They are building great stuff and helping people of all skill-levels learn how to use it.

Everyone in this workshop is coming from a different place with different experiences and expectations. But everyone will learn something new here, because there is so much innovation in the data science world. Even instructors and helpers learn something new every time, from each other and from your questions. You are all welcome here and encouraged to help each other.

Here are some important themes throughout:

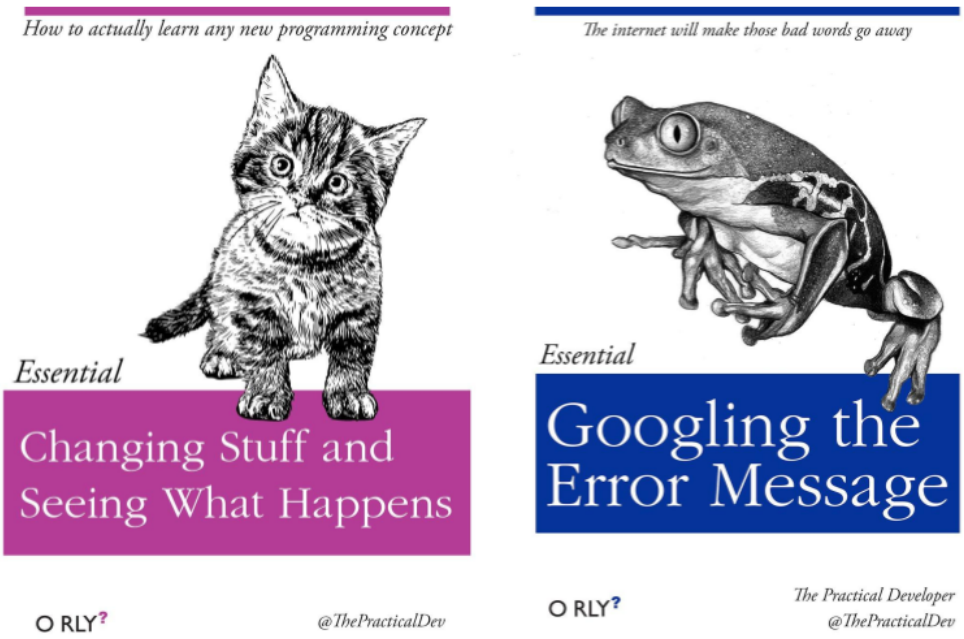


Figure 2.1:

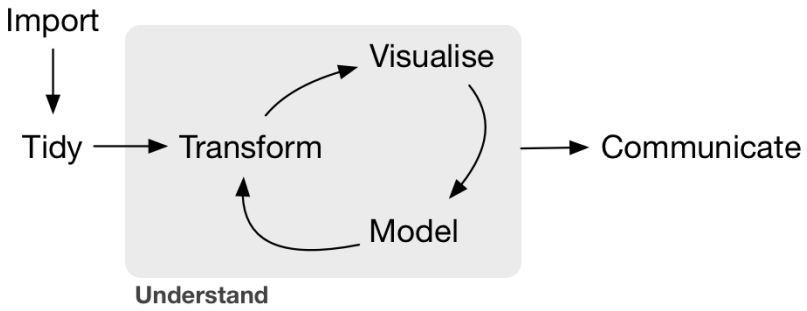


Figure 2.2:

2.1.1 Tidy data workflow

We will be learning about tidy data.

Hadley Wickham has developed a ton of the tools we'll use today. Here's an overview of techniques to be covered in Hadley Wickham and Garrett Grolemund of RStudio's book R for Data Science:

We will be focusing on:

- **Tidy**: `tidyr` to organize rows of data into unique values
- **Transform**: `dplyr` to manipulate/wrangle data based on subsetting by rows or columns, sorting and joining
- **Visualise**:
  - `ggplot2` static plots, using grammar of graphics principles
- **Communicate**
  - online website with *Github Pages*
  - version with *git*

  - dynamic documents with *Rmarkdown*

2.2 Gapminder data:

We'll be using the gapminder dataset pioneered by Hans Rosling. These data represent the health and wealth of every nation in the world.

While these data are not conservation or environmental oriented, it is a fantastically rich data set with many parallels to data you may have and wrangling you will need to do. It's important to be open to separate your science questions from data questions, and working with other people's data is a good way to do it. These data will be familiar to data that you're likely working with: there is information for many indicators for many study sites for many years.

2.3 By the end of the course...

By the end of the course you'll wrangle the gapminder data, make your own graphics that you'll publish on a webpage you've built with GitHub and RMarkdown. Woop!

I made this training book with GitHub and RStudio's RMarkdown, which is what we'll be learning in the workshop.

2.4 Prerequisites

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

1. Have up-to-date versions of `R` and RStudio and have RStudio configured with Git/GitHub
    - Download and install R: https://cloud.r-project.org
    - Download and install RStudio: http://www.rstudio.com/download
    - Create a GitHub account: https://github.com *Note! Shorter names that kind of identify you are better, and use your work email!*
2. Get comfortable: if you're not in a physical workshop, be set up with two screens if possible. You will be following along in RStudio on your own computer while also watching a virtual training or following this tutorial on your own.

2.5 Credit

This material builds from a lot of fantastic materials developed by others in the open data science community. In particular, it pulls from the following resources, which are highly recommended for further learning and as resources later on. Specific lessons will also cite more resources.

- R for Data Science by Hadley Wickham and Garrett Grolemund
- STAT 545 by Jenny Bryan
- Happy Git with R by Jenny Bryan
- Software Carpentry by the Carpentries

Chapter 3

R & RStudio, Rmarkdown

3.1 Overview

Objectives

In this lesson we will:

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

Resources

This lesson is a combination of excellent lessons by others (thank you Jenny Bryan and Data Carpentry!) that I have combined and modified for our workshop today. I definitely recommend reading through the original lessons and using them as reference:

Dr. Jenny Bryan's lectures from STAT545 at UBC

- R basics, workspace and working directory, RStudio projects
- Basic care and feeding of data in R

RStudio has great resources about its IDE (IDE stands for integrated development environment):

- webinars
- cheatsheets

3.2 Why learn R with RStudio

You are all here today to learn how to code. Coding made me a better scientist because I was able to think more clearly about analyses, and become more efficient in doing so. Data scientists are creating tools that make coding more intuitive for new coders like us, and there is a wealth of awesome instruction and resources available to learn more and get help.

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

And **if R were an airplane, RStudio is the airport**. RStudio provides support! Runways, communication, community, and other services, and just makes your overall life easier. So it's not just the infrastructure

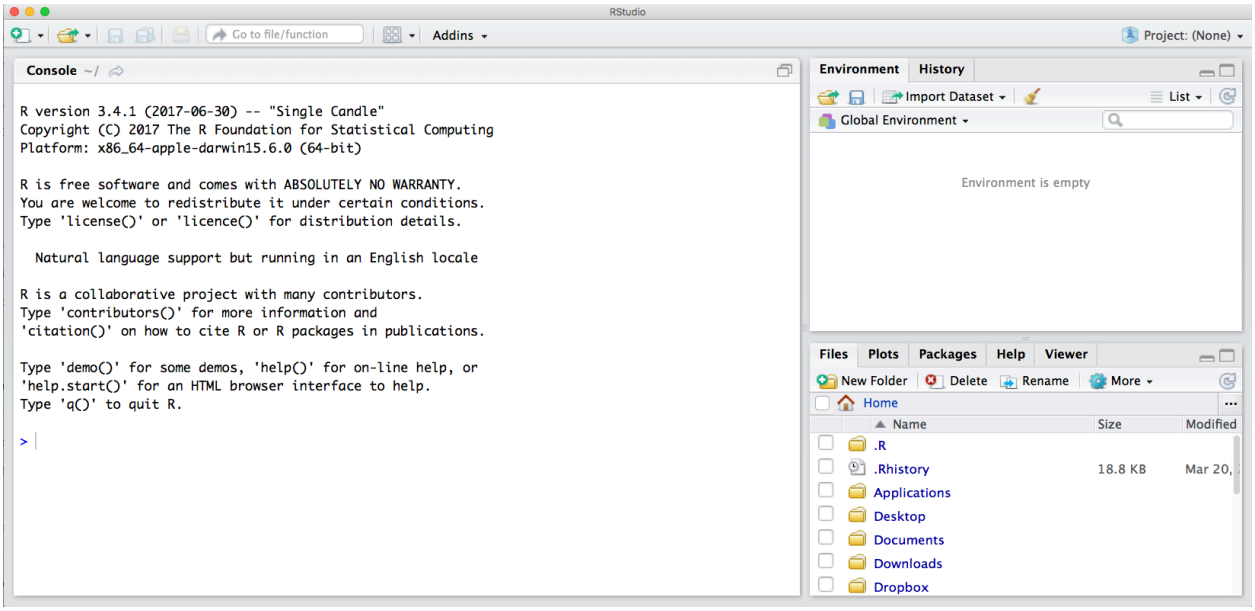


Figure 3.1:

(the user interface or IDE), although it is a great way to learn and interact with your variables, files, and interact directly with GitHub. It's also data science philosophy, R packages, community, and more. So although you can fly your plane without an airport and we could learn R without RStudio, that's not what we're going to do.

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

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

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

3.3 R at the console, RStudio goodies

Launch RStudio/R.

Notice the default panes:

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

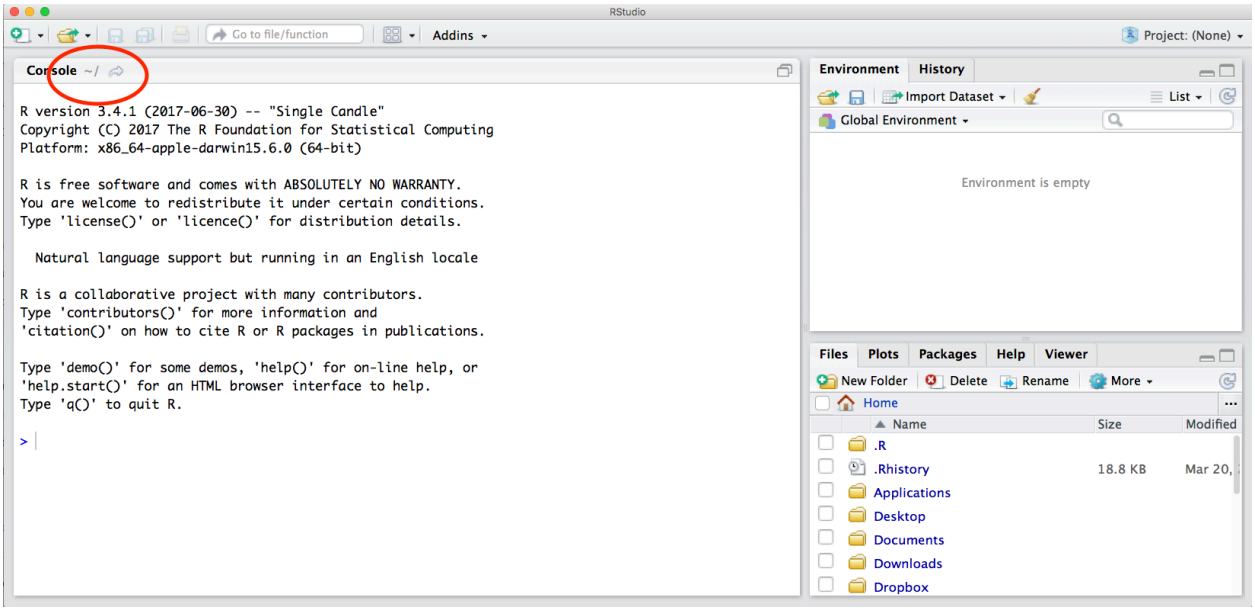


Figure 3.2:

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

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

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

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

Make an assignment and then inspect the object you just created.

```
x <- 3 * 4
x
```

```
## [1] 12
```

In my head I hear, e.g., "x gets 12".

All R statements where you create objects – "assignments" – have this form: `objectName <- value`.

I'll write it in the command line with a hashtag `#`, which is the way R comments so it won't be evaluated.

```
## objectName <- value

## This is also how you write notes in your code to explain what you are doing.
```

Object names cannot start with a digit and cannot contain certain other characters such as a comma or a space. You will be wise to adopt a convention for demarcating words in names.

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

Make an assignment

```
this_is_a_really_long_name <- 2.5
```

To inspect this variable, instead of typing it, we can press the up arrow key and call your command history, with the most recent commands first. Let's do that, and then delete the assignment:

```
this_is_a_really_long_name
```

```
## [1] 2.5
```

Another way to inspect this variable is to begin typing `this_`…and RStudio will automagically have suggested completions for you that you can select by hitting the tab key, then press return.

One more:

```
science_rocks <- 100
```

Let's try to inspect:

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

3.3.1 Error messages are your friends

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

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

And also know that there are errors that can creep in more subtly, when you are giving information that is understood, but not in the way you meant. Like if I'm telling a story about tables and you're picturing where you eat breakfast and I'm talking about data. This can leave me thinking I've gotten something across that the listener (or R) interpreted very differently. And as I continue telling my story you get more and more confused… So write clean code and check your work as you go to minimize these circumstances!

3.3.2 Logical operators and expressions

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

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

```
science_rocks == 2
```

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

```
science_rocks <= 30
```

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

```
science_rocks != 5
```

```
## [1] TRUE
```

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

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

When assigning a value to an object, R does not print anything. You can force R to print the value by using parentheses or by typing the object name:

```
weight_kg <- 55    # doesn't print anything
(weight_kg <- 55)  # but putting parenthesis around the call prints the value of `weight_kg`
```

```
## [1] 55
```

```
weight_kg          # and so does typing the name of the object
```

```
## [1] 55
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight into pounds (weight in pounds is 2.2 times the weight in kg):

```
2.2 * weight_kg
```

```
## [1] 121
```

We can also change a variable's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

```
## [1] 126.5
```

This means that assigning a value to one variable does not change the values of other variables. For example, let's store the animal's weight in pounds in a new variable, `weight_lb`:

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`? 126.5 or 220? Why?

3.4 R functions, help pages

R has a mind-blowing collection of built-in functions that are used with the same syntax: function name with parentheses around what the function needs in order to do what it was built to do. When you type a function like this, we say we are "calling the function". `verb(noun = something, adjective = something, etc)`. This example is from R for Data Science using a children's poem called Little Bunny Foo Foo.

We can call a function without passing it anything (nothing inside the closed parentheses), and assign it to a variable called `foo_foo`.

```
## foo_foo <- little_bunny()
```

And since `foo_foo` is an object, you can pass it to other functions:

```
## hop(foo_foo, through = forest)
## scoop(foo_foo, up = field_mice)
## bop(foo_foo, on = head)
```

What would happen if I tried to run one of those lines above? I would get an error because they aren't real functions, and R tells me so:

```
foo_foo <- little_bunny()
# Error in little_bunny() : could not find function "little_bunny"
```

And that's great, this error message is helpful: R doesn't know what the `little_bunny` function is, and to be honest, neither do we. We didn't expect that it would know what to do. OK, so now let's look at a real function.

Let's try using `seq()` which makes regular sequences of numbers and, while we're at it, demo more helpful features of RStudio.

Type `se` and hit TAB. A pop up shows you possible completions. Specify `seq()` by typing more to disambiguate or using the up/down arrows to select. Notice the floating tool-tip-type help that pops up, reminding you of a function's arguments. If you want even more help, press F1 as directed to get the full documentation in the help tab of the lower right pane.

Type the arguments `1, 10` and hit return.

```
seq(1, 10)
```

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

We could probably infer that the `seq()` function makes a sequence, but let's learn for sure. Type (and you can autocomplete) and let's explore the help page:

```
?seq
help(seq) # same as ?seq
```

```
seq(from = 1, to = 10) # same as seq(1, 10); R assumes by position
```

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

```
seq(from = 1, to = 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

The above also demonstrates something about how R resolves function arguments. You can always specify in `name = value` form. But if you do not, R attempts to resolve by position. So above, it is assumed that we want a sequence `from = 1` that goes `to = 10`. Since we didn't specify step size, the default value of `by` in the function definition is used, which ends up being 1 in this case. For functions I call often, I might use this resolve by position for the first argument or maybe the first two. After that, I always use `name = value`.

The help page tells the name of the package in the top left, and broken down into sections:

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

The examples can be copy-pasted into the console for you to understand what's going on. Remember we were talking about expecting there to be a function for something you want to do? Let's try it.

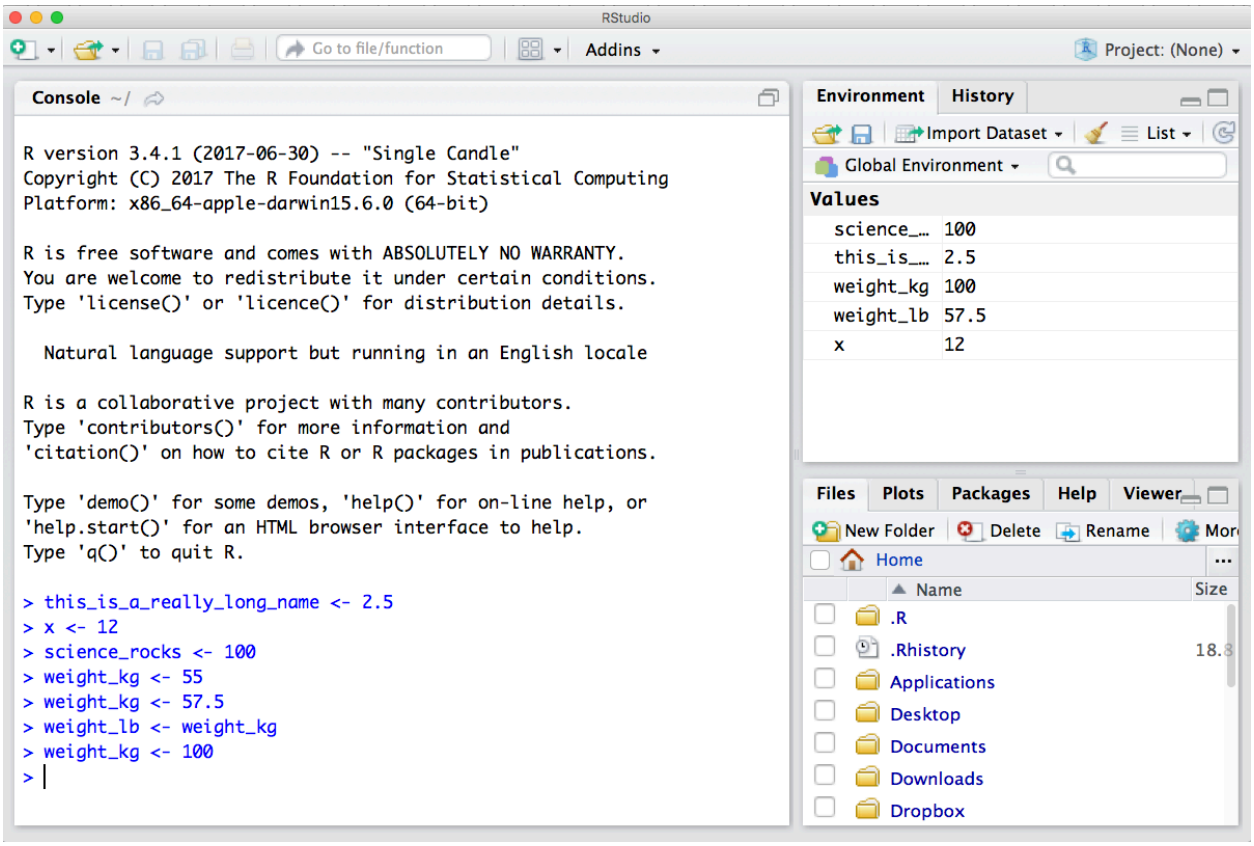


Figure 3.3:

3.4.1 Your turn

> Exercise: Talk to your neighbor(s) and look up the help file for a function that you know or expect to exist. Here are some ideas: `?getwd()`, `?plot()`, `min()`, `max()`, `?mean()`, `?log()`).

And there's also help for when you only sort of remember the function name: double-questionmark:

```
??install
```

As we saw with creating the `foo_foo` variable above, not all functions have (or require) arguments:

```
date()
```

```
## [1] "Mon Nov 27 19:13:26 2017"
```

3.5 Clearing the environment

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

You can also get a listing of these objects with a few different R commands:

```
objects()
```

```
## [1] "science_rocks"             "this_is_a_really_long_name"
## [3] "weight_kg"                 "weight_lb"
## [5] "x"
```

```
ls()
```

```
## [1] "science_rocks"              "this_is_a_really_long_name"
## [3] "weight_kg"                  "weight_lb"
## [5] "x"
```

If you want to remove the object named `weight_kg`, you can do this:

```
rm(weight_kg)
```

To remove everything:

```
rm(list = ls())
```

or click the broom in RStudio's Environment pane.

3.5.1 Your turn

> Exercise: Clear your workspace, then create a few new variables. Create a variable that is the mean of a sequence of 1-20. What's a good name for your variable? Does it matter what your 'by' argument is? Why?

3.6 RMarkdown

Now we are going to also introduce RMarkdown. This is really key for collaborative research, so we're going to get started with it early and then use it for the rest of the day.

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

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

You can give it a Title like "My Project". Then click OK.

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

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

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

Let's go ahead and "Knit HTML".

What do you notice between the two?

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

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

More details…

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

We don't know that much R yet, but you can see that we are taking a summary of some data called 'cars', and then plotting. There's a lot more to learn about R, and we'll get into it for the next few days.

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

Notice the syntax for:

```
---
title: "My Project"
author: "Julie"
date: "11/21/2017"
output: html_document
---

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

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```{r cars}
summary(cars)
```

Including Plots

You can also embed plots, for example:

```{r pressure, echo=FALSE}
plot(pressure)
```

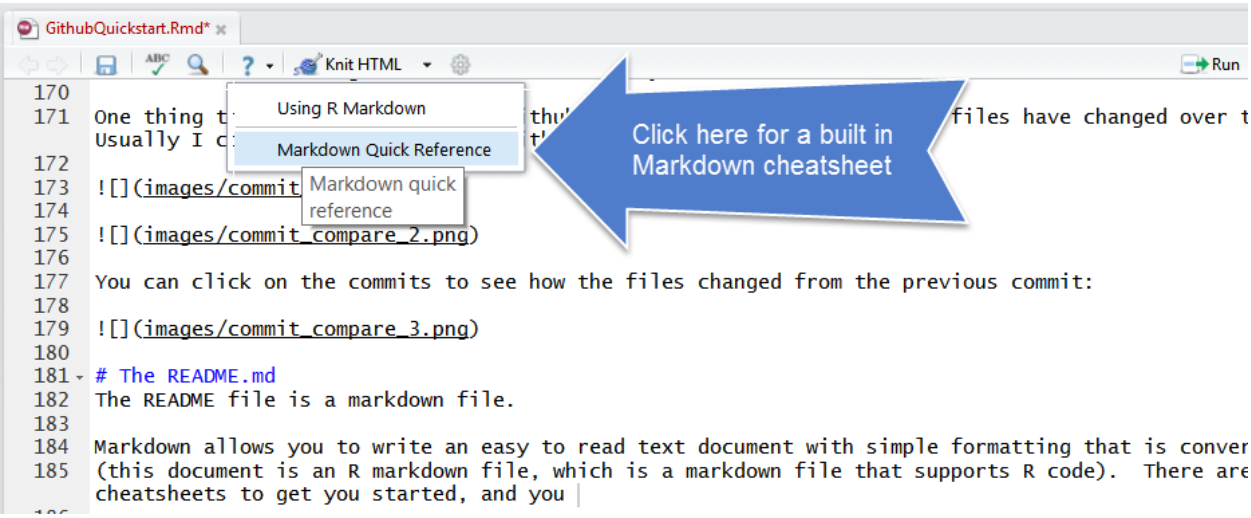
Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.
```

Figure 3.4:

Figure 3.5:

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

There are some good cheatsheets to get you started, and here is one built into RStudio:



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

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

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

3.6.1 Your Turn

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

3.7 Troubleshooting

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

3.7.1 I just entered a command and nothing's happening

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

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

3.7.2 How do I update RStudio?

To see if you have the most current version of RStudio, go to the Help bar > Check for Updates. If there is an update available, you'll have the option to Quit and Download, which will take you to http://www.rstudio.com/download. When you download and install, choose to replace the previous version.

Chapter 4

GitHub

We will learn about version control using git and GitHub, and we will interface with this through RStudio.

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

4.1 Objectives & Resources

4.1.0.1 Objectives

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

Here's what we'll do (we already set up git on our local computer in the previous section):

1. create a repository on Github.com
2. clone locally using RStudio
3. learn the RStudio-GitHub workflow by syncing to Github.com: pull, stage, commit, push
4. explore github.com: files, commit history, file history
5. practice the RStudio-GitHub workflow by editing and adding files
6. practice R Markdown

4.1.0.2 Resources

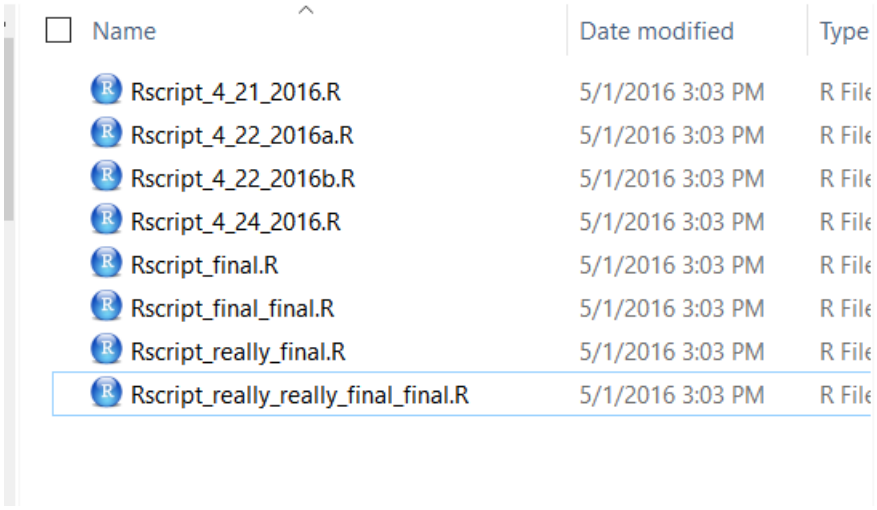
These materials borrow from:

- Jenny Bryan's lectures from STAT545 at UBC: The Shell
- Jenny Bryan's Happy git with R tutorial
- Melanie Frazier's GitHub Quickstart
- Ben Best's Software Carpentry at UCSB

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

4.2 Why should scientists use Github?

1. Ends (or, nearly ends) the horror of keeping track of versions. Basically, we get away from this:



   When you open your repository, you only see the most recent version. But, it easy to compare versions, and you can easily revert to previous versions.
2. Improves collaborative efforts. Different researchers can work on the same files at the same time!
3. It is easy to share and distribute files through the Github website.
4. Your files are available anywhere, you just need internet connection!

4.2.1 What are Git and Github?

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

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

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

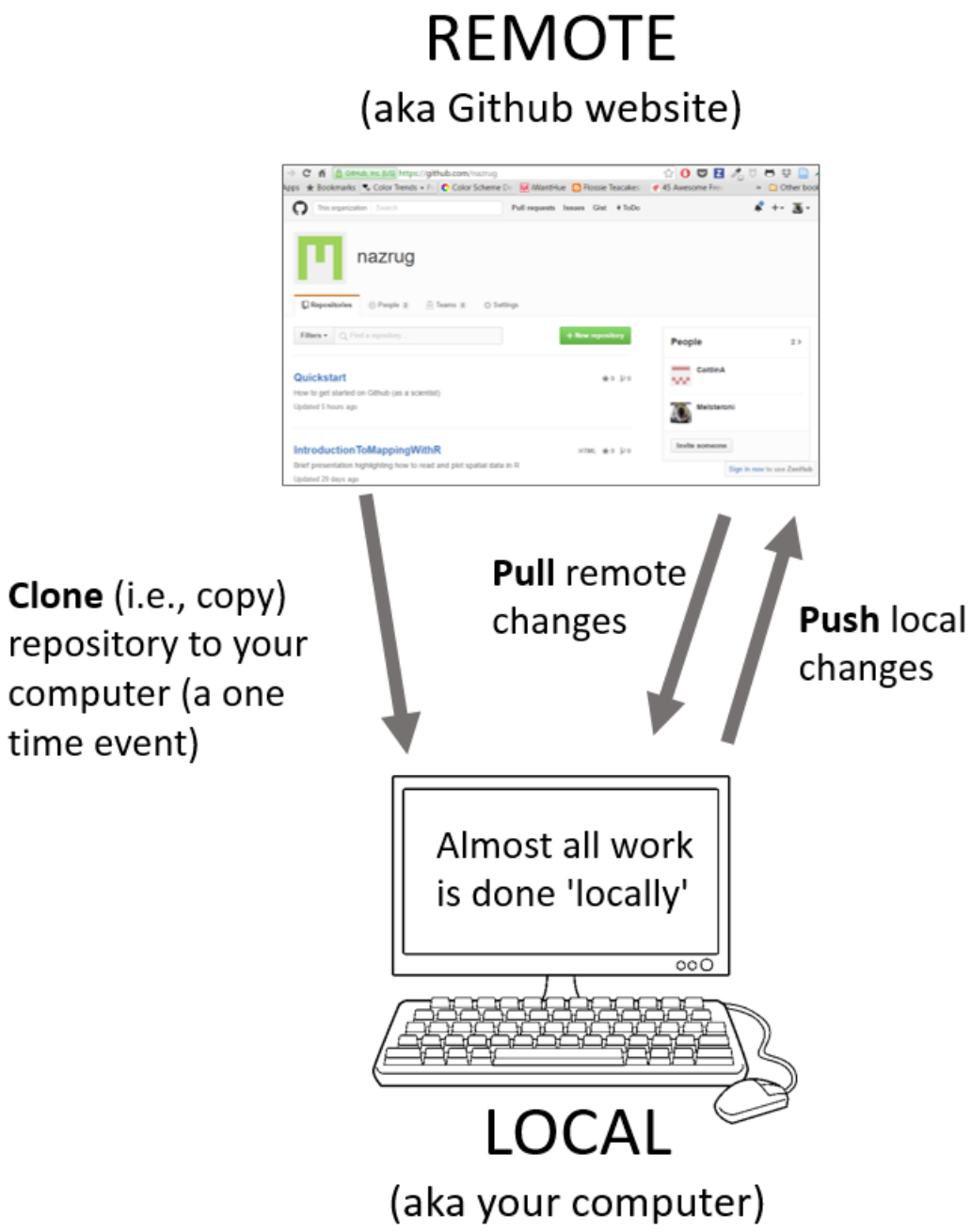
These concepts are more important for coders who want the entire coding community (and not just people working on the same project) to be able to suggest changes to their code. This isn't how most scientists will use Github.

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

4.2.2 Some Github terminology

- **User**: A Github account for you (e.g., jules32).

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



4.3 Setup Git & GitHub

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

1. Create **Github** account at http://github.com, if you don't already have one. For username, I recommend all lower-case letters, short as you can. I recommend using your *.edu email*, since you can request free private repositories via GitHub Education discount.

2. Configure **git** with global commands, which means it will apply 'globally' to all files on your computer, rather than to a specific folder. Open the Git Bash program (Windows) or the Terminal (Mac) and type the following:

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

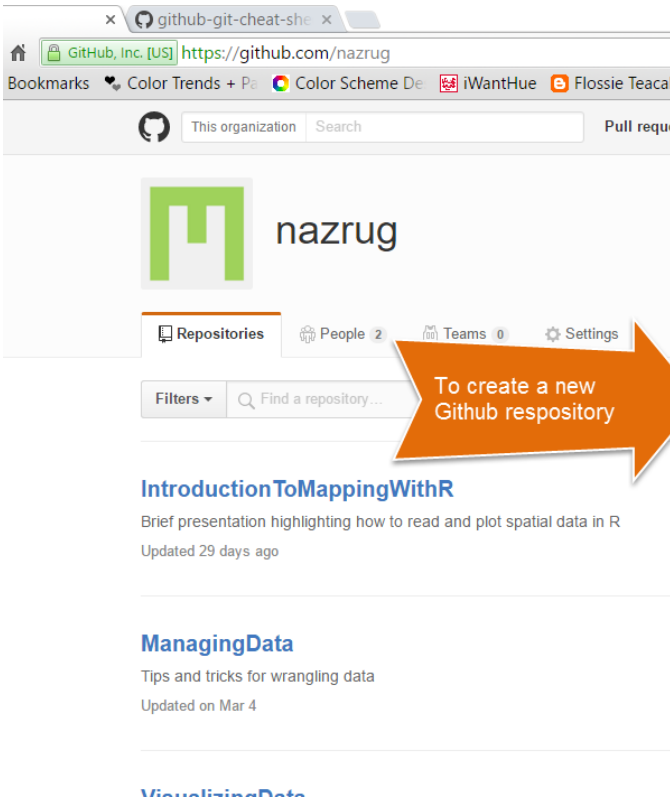
# replace USER with your Github user account
git config --global user.name USER

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

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

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

4.4 Create a repository on Github.com



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

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

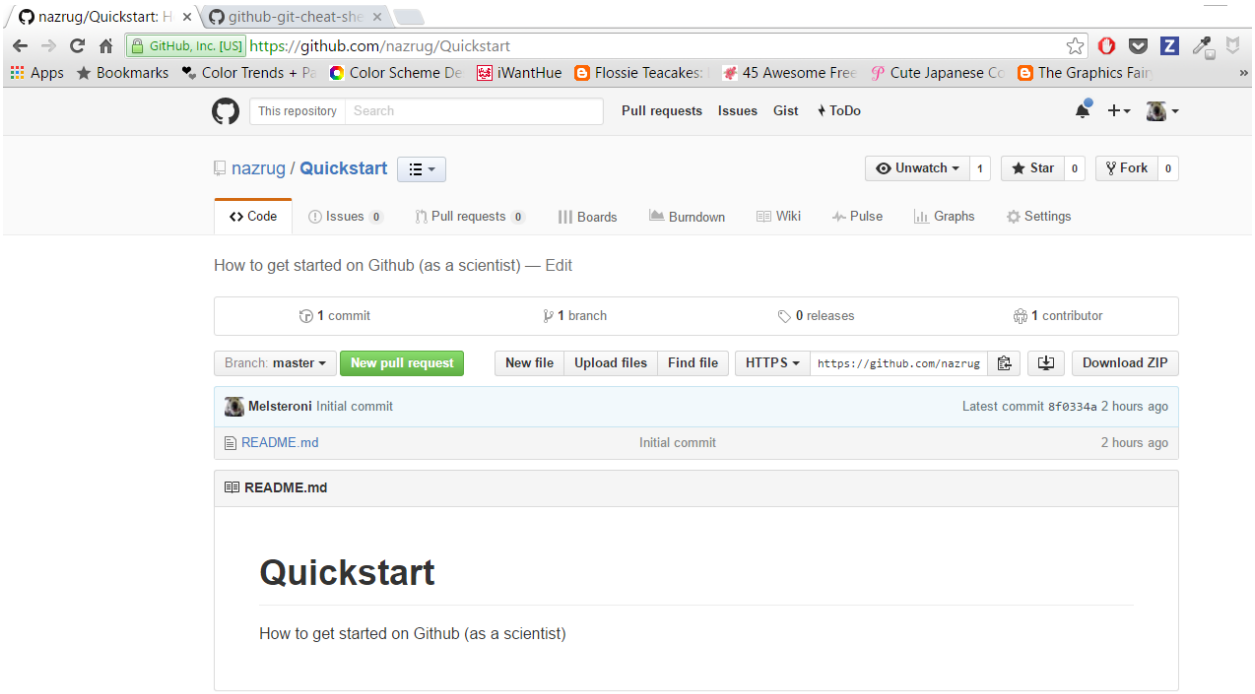
Also, add a description, make it public, create a README file, and create your repo!

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

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

Check out our new repository!

Notice how the README.md file we created is automatically displayed at the bottom.



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

4.5 Clone your repository using RStudio

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

Step 0: Create your `github` folder

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

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

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

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

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

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

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

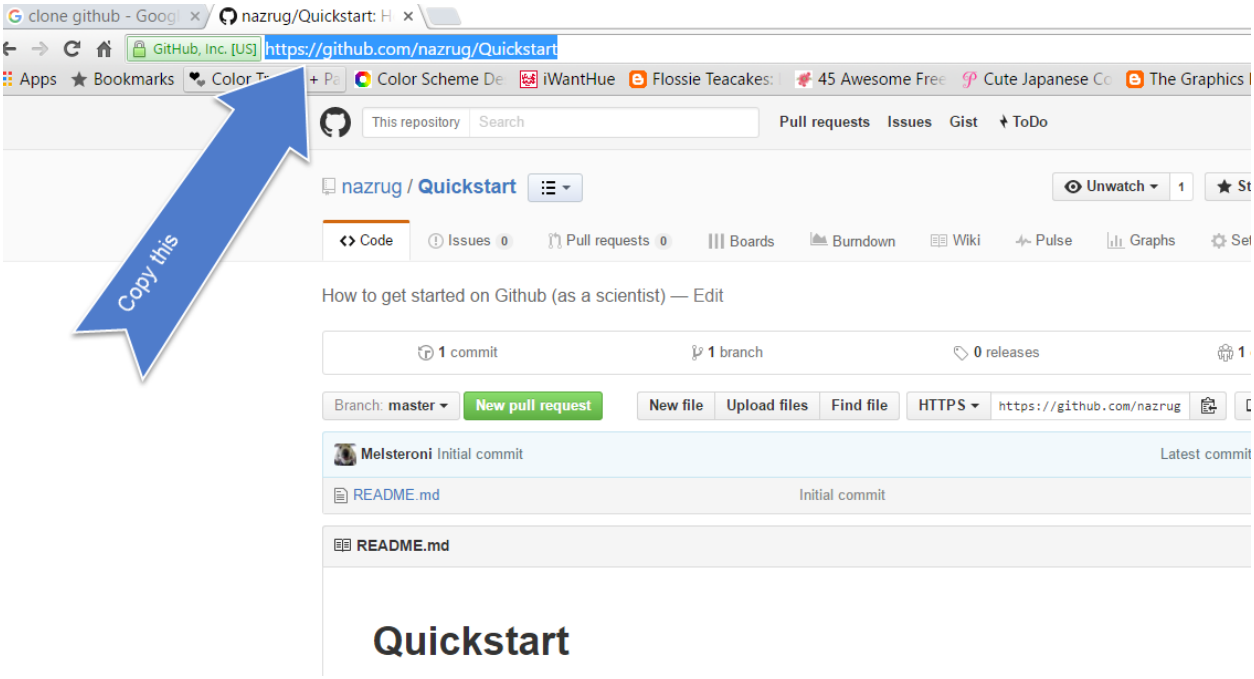
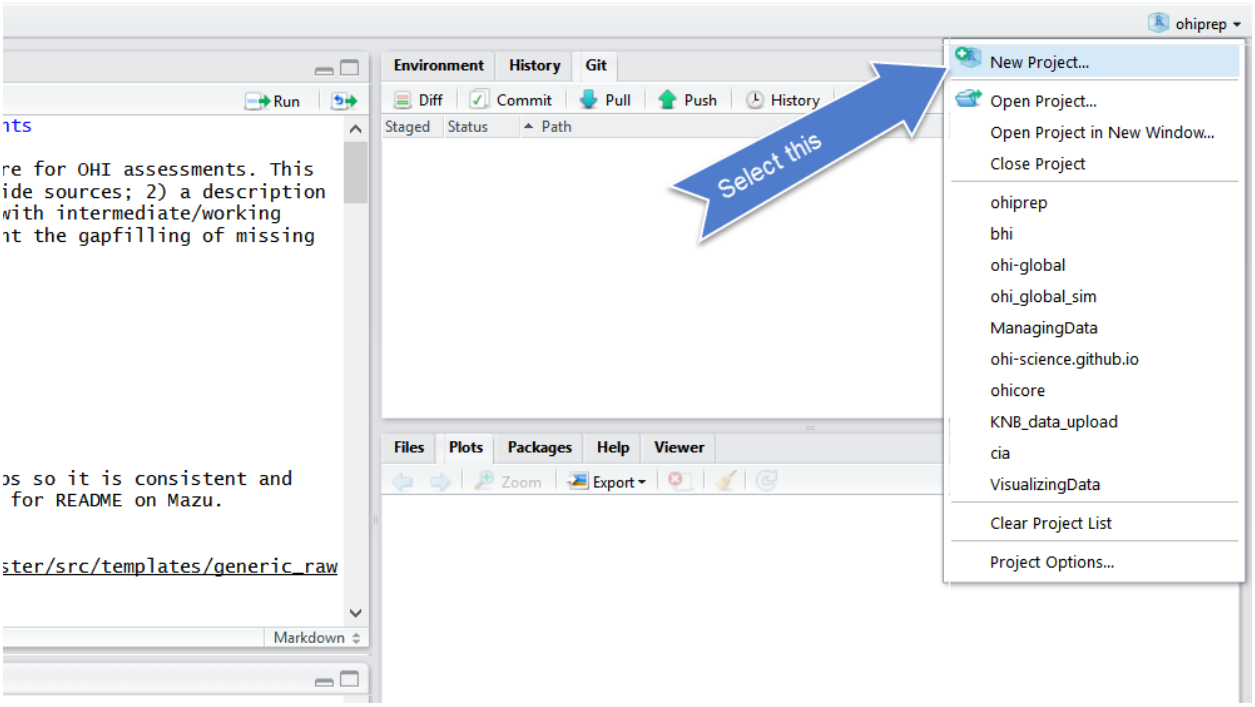
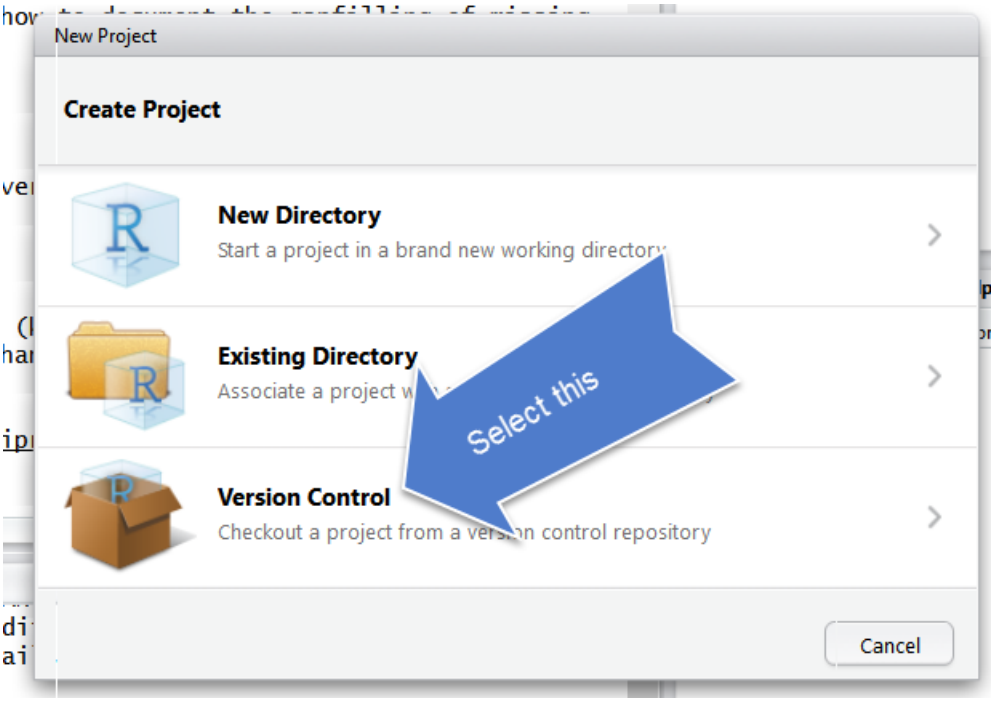


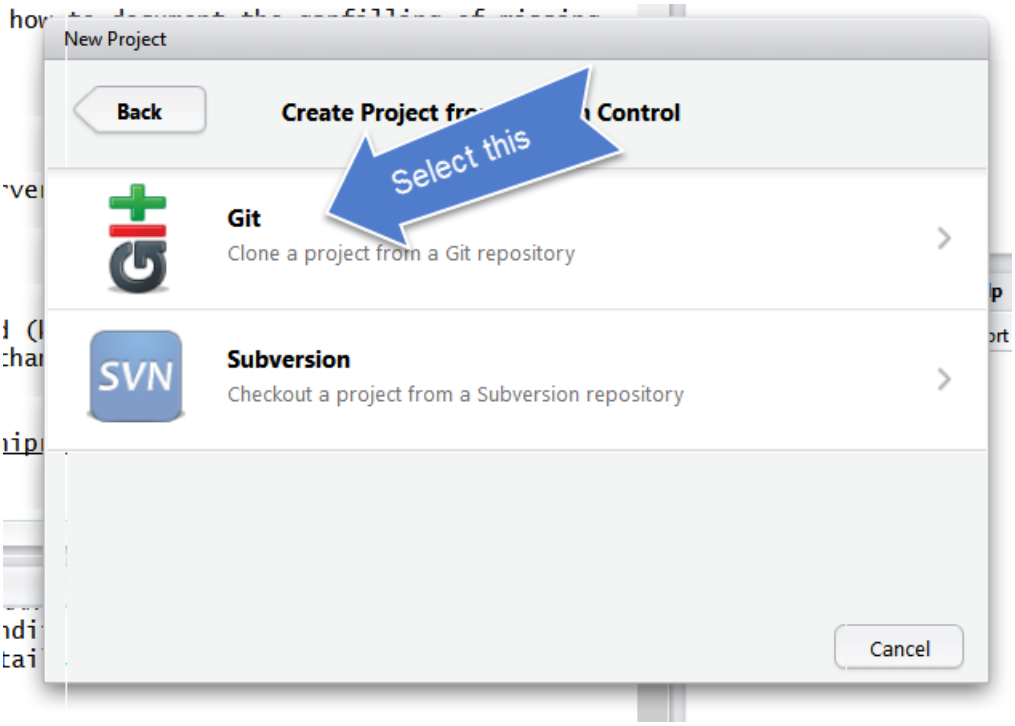
Figure 4.1:



Step 3: Select Version Control



Step 4: Select Git



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

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

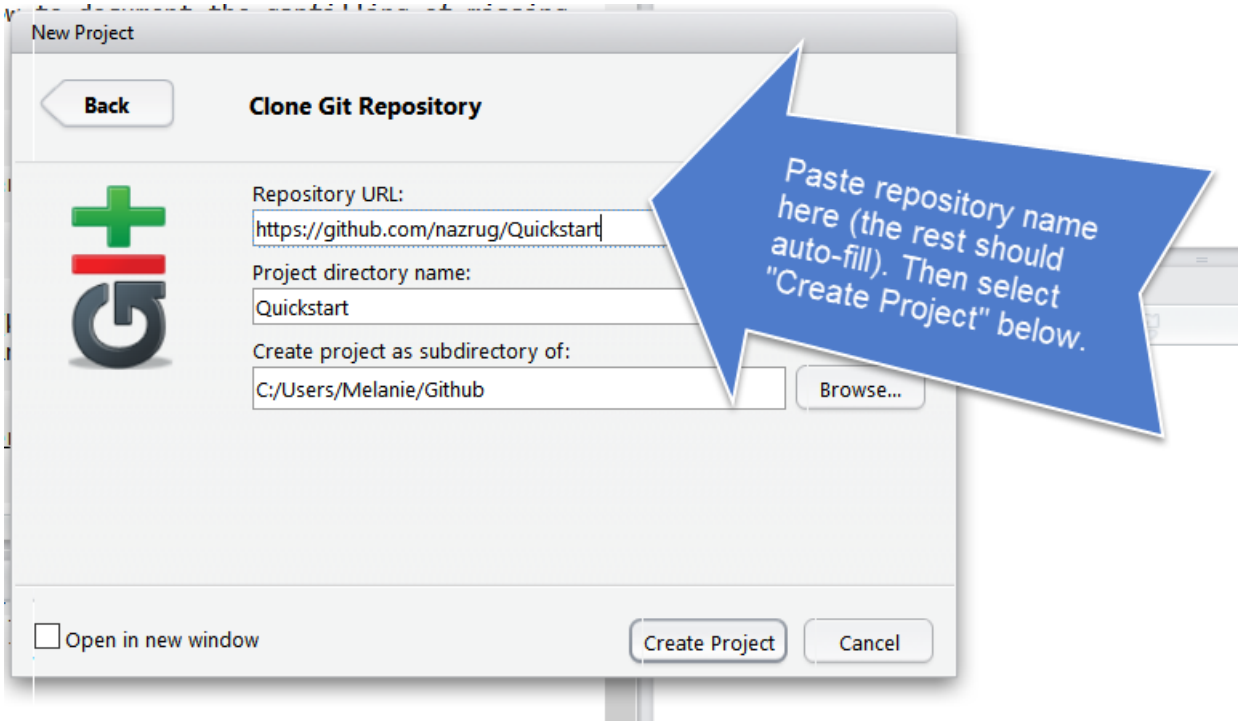
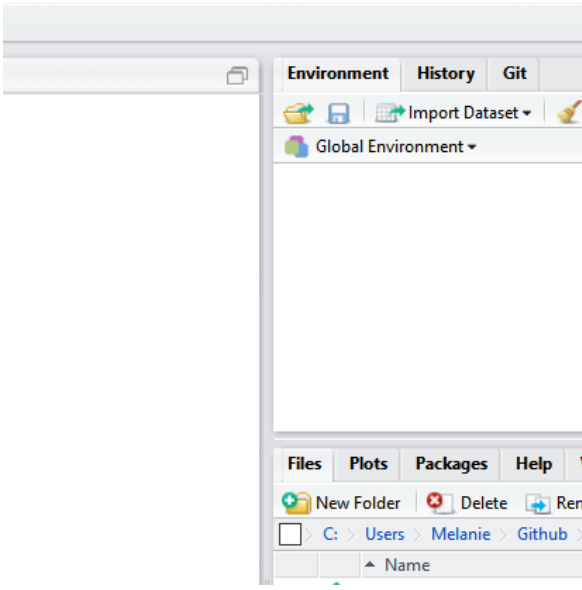
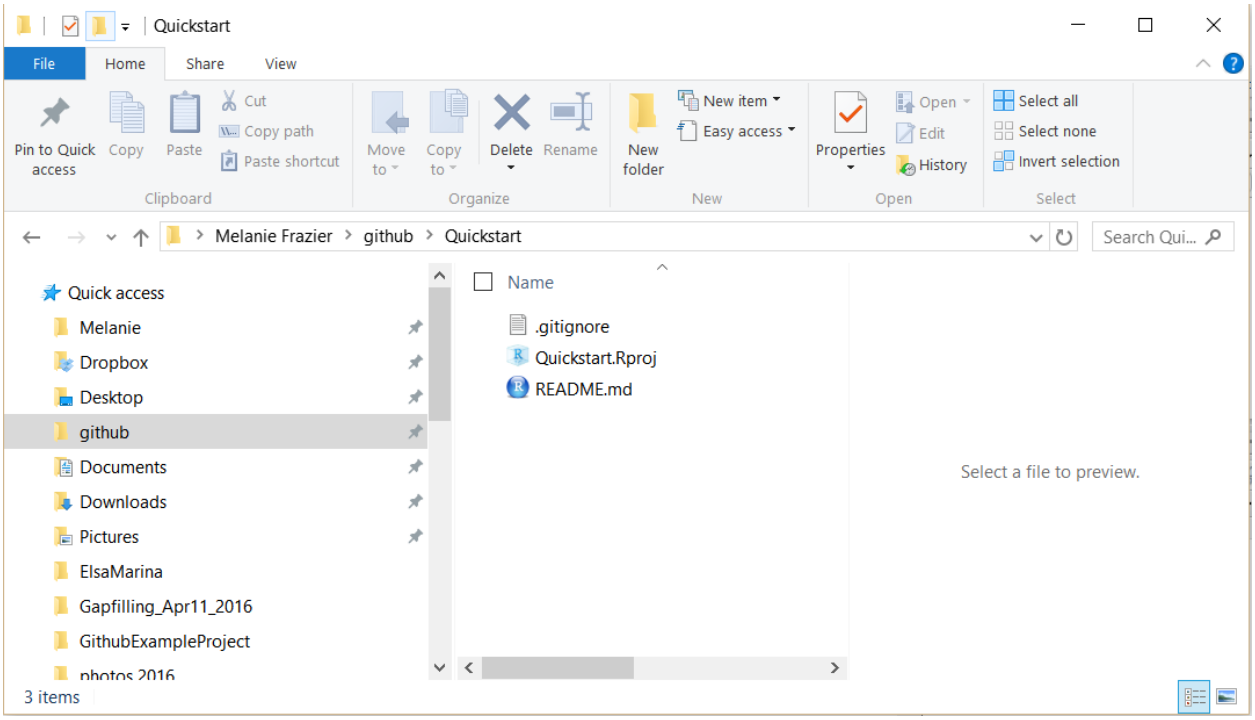


Figure 4.2:



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

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



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

4.6 Inspect your repository

Notice a few things in our repo here:

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

4.7 Add files to our local repo

The repository will contain:

- .gitignore file
- README.md
- Rproj

And, I typically create the following:

- folders for "data" and "figures"

- R scripts
- etc.

I'm going to copy-paste a small from my desktop into the folder.

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

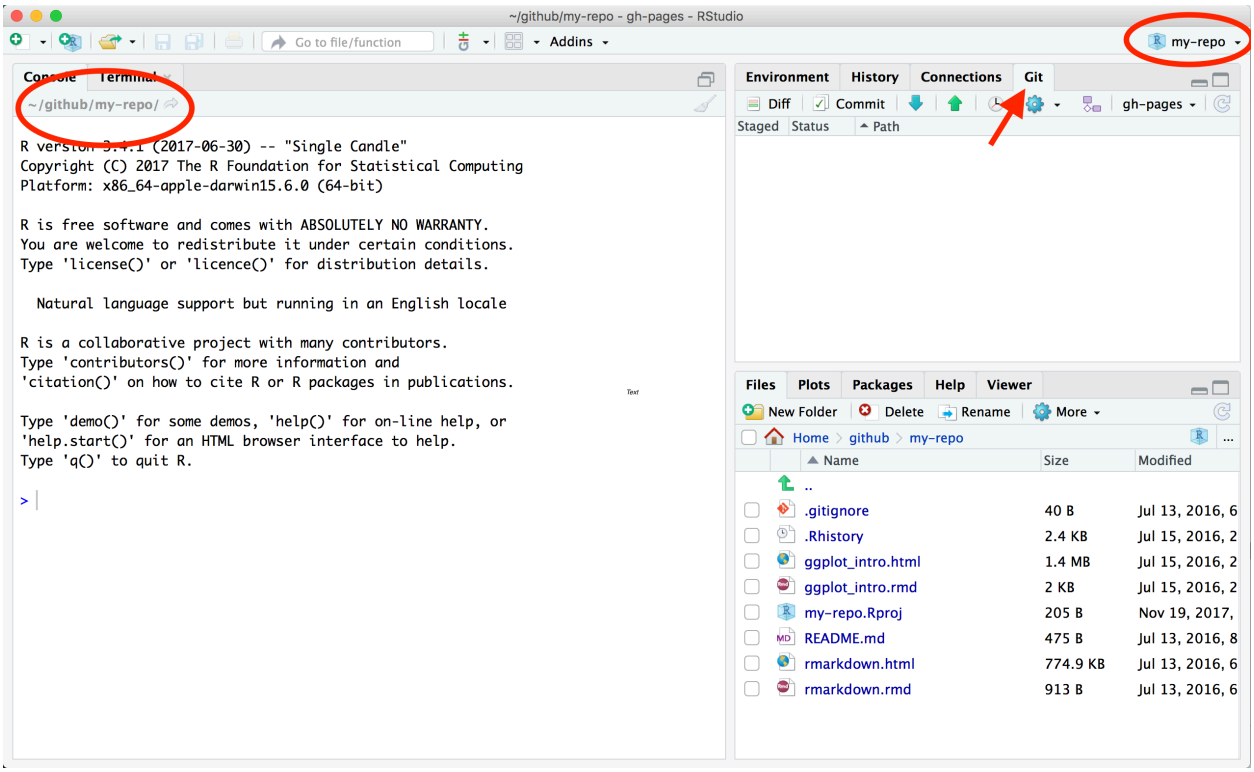
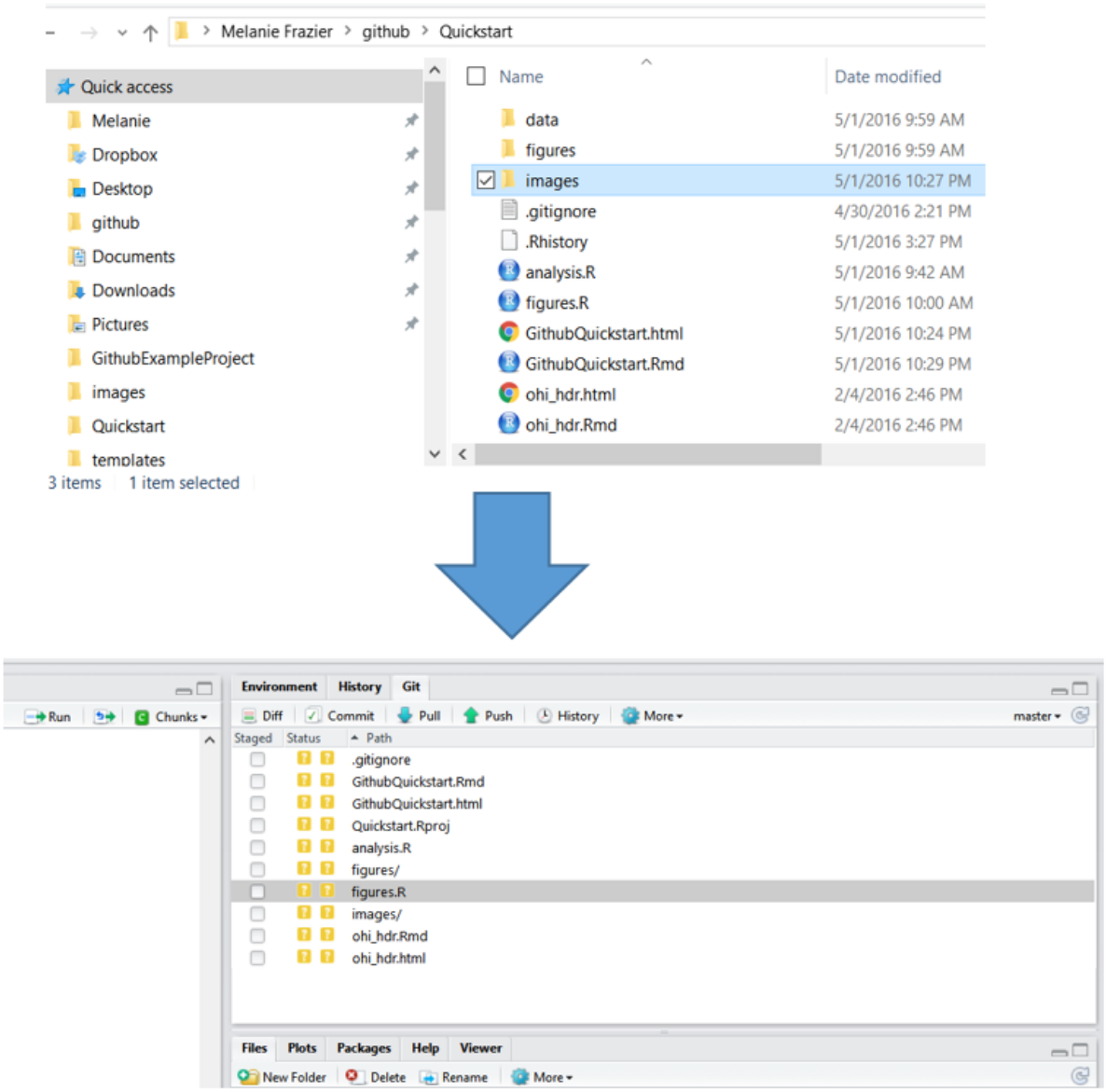


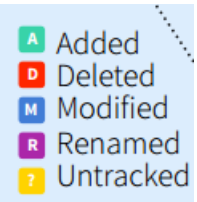
Figure 4.3:

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



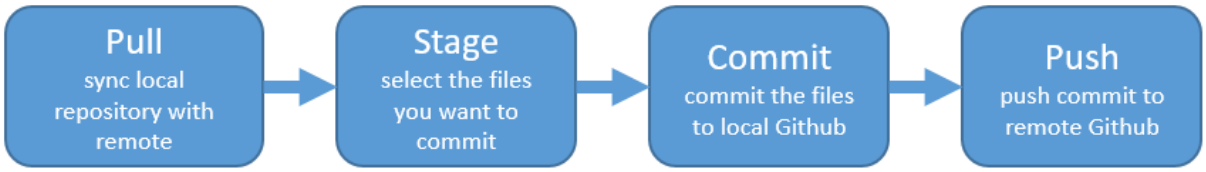
4.7.1 Inspect what has changed

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



4.8 Sync from RStudio to GitHub

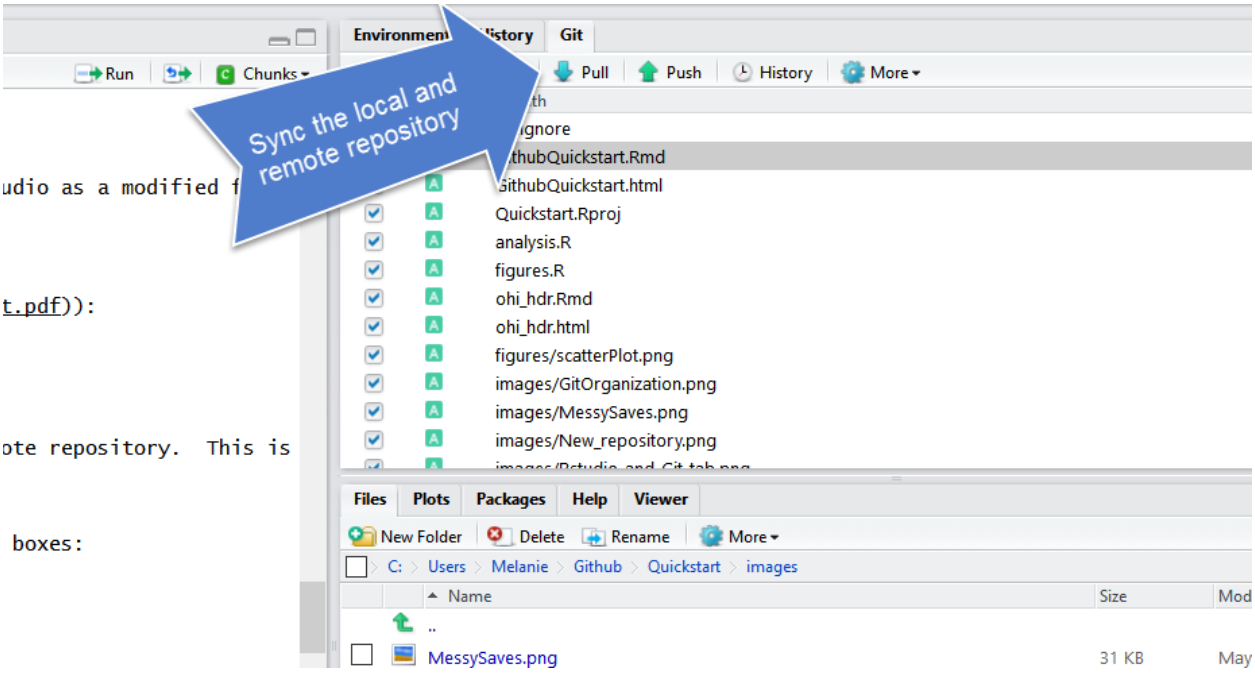
When you are ready to commit your changes, you follow these steps:



We walk through this process below:

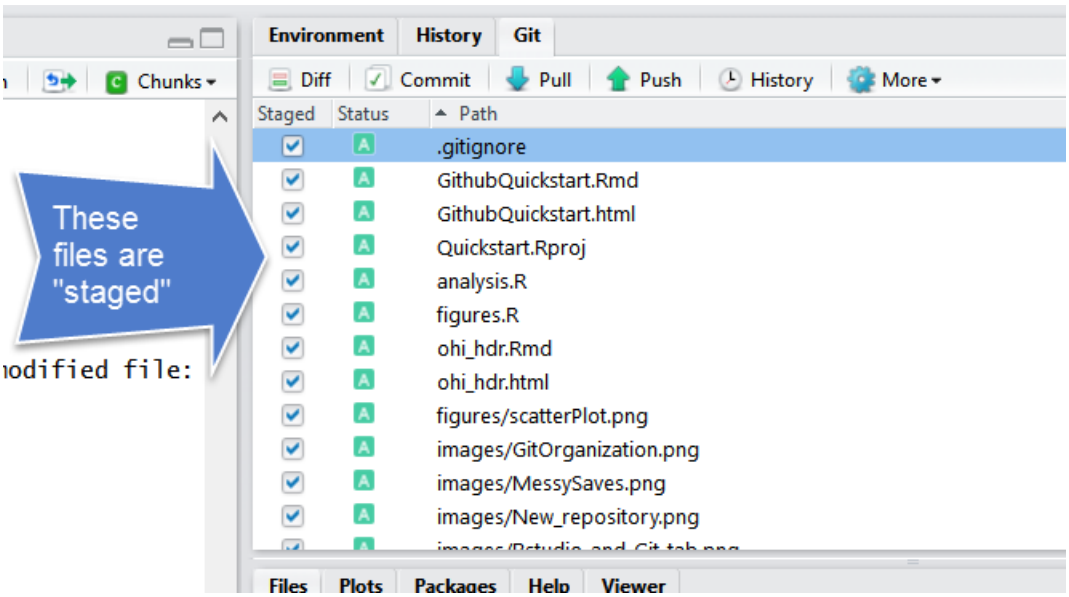
4.8.1 Pull

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

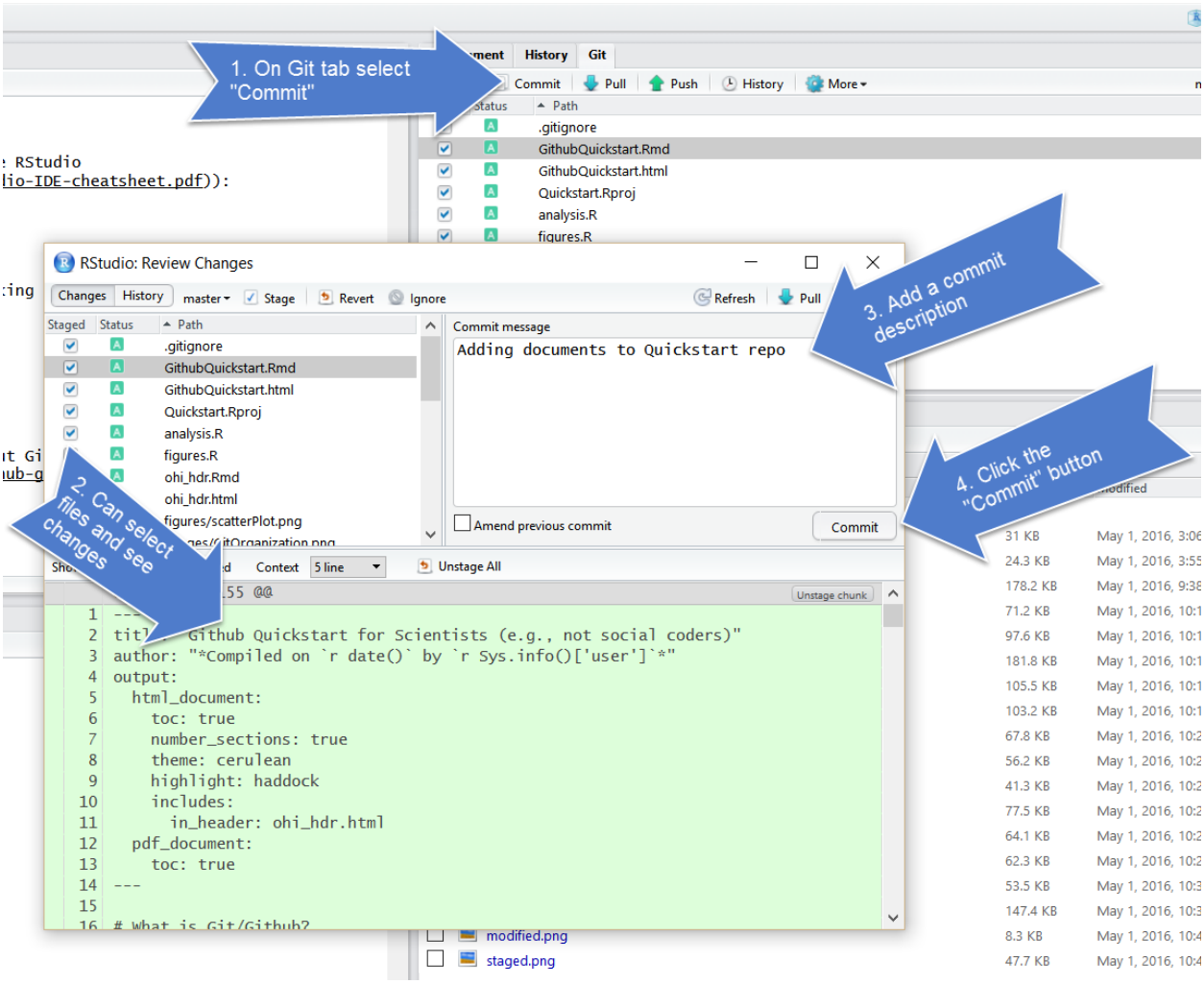


4.8.2 Stage

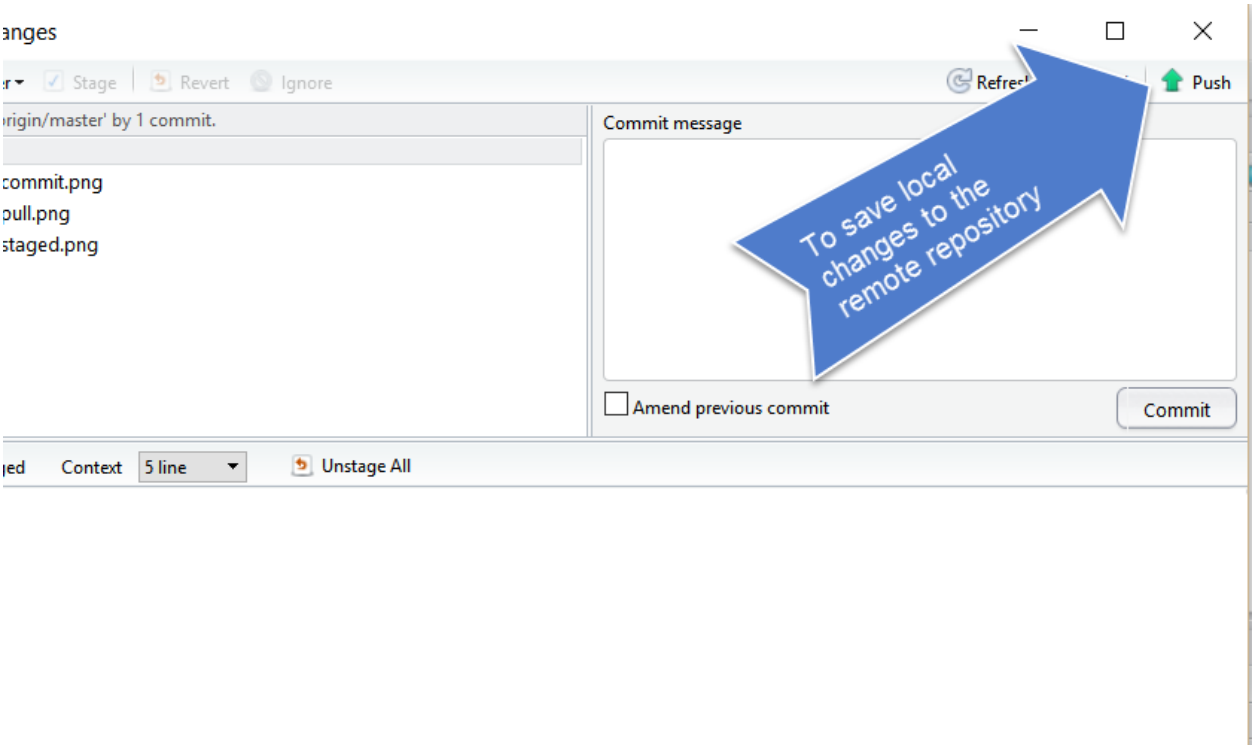
Stage the files you want to commit. In RStudio, this involves checking the "Staged" boxes:


Environment
History
Git
Chunks
Diff
Commit
Pull
Push
History
More
Staged
Status
Path
.gitignore
GithubQuickstart.Rmd
GithubQuickstart.html
Quickstart.Rproj
analysis.R
figures.R
ohi_hdr.Rmd
ohi_hdr.html
figures/scatterPlot.png
images/GitOrganization.png
images/MessySaves.png
images/New_repository.png
These files are "staged"
Files
Plots
Packages
Help
Viewer

4.8.3 Commit

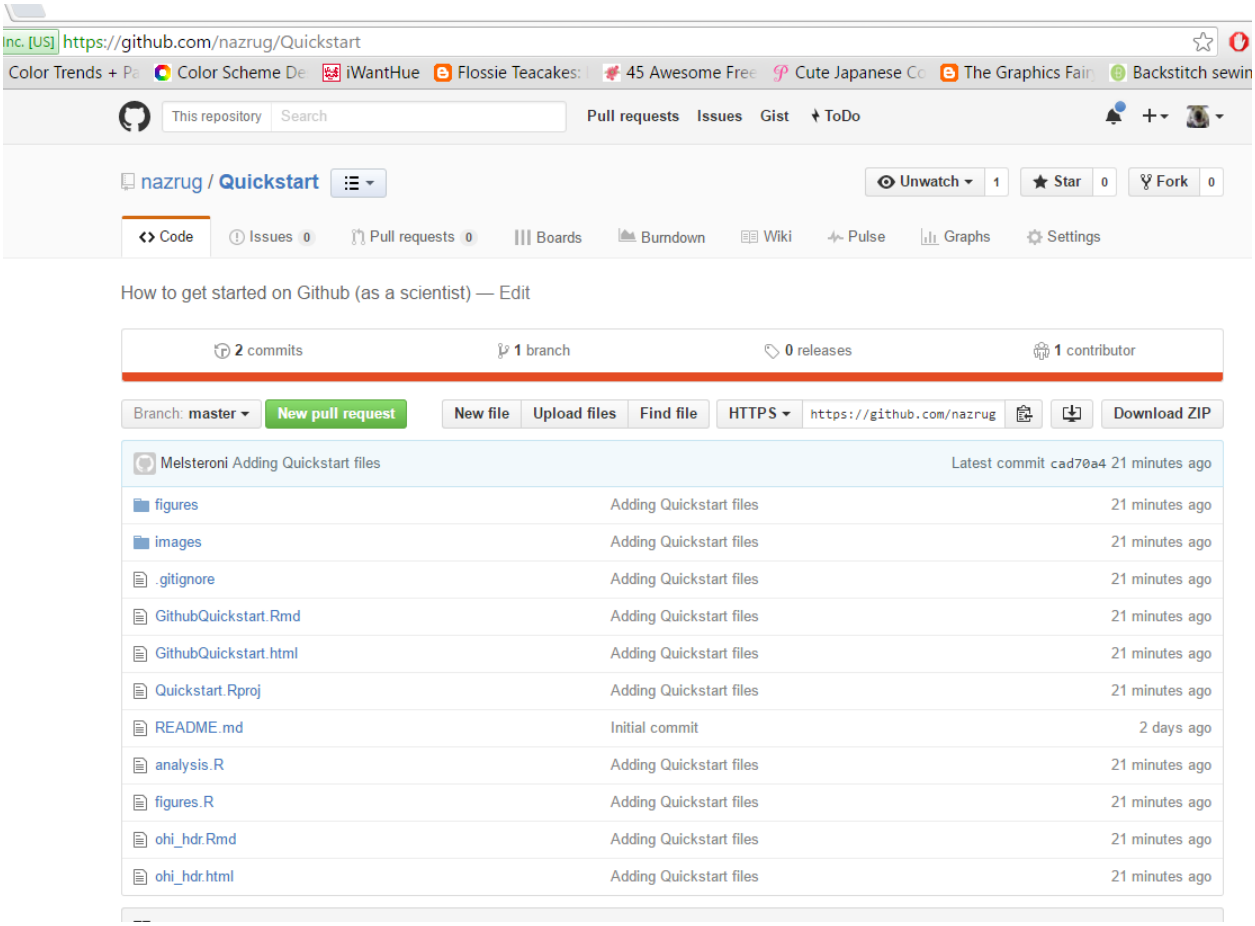

1. On Git tab select "Commit"
RStudio: Review Changes
Changes
History
master
Stage
Revert
Ignore
Refresh
Pull
Commit message
Adding documents to Quickstart repo
3. Add a commit description
2. Can select files and see changes
4. Click the "Commit" button
Amend previous commit
Commit
Unstage All
Unstage chunk
Context
5 line
Github Quickstart for Scientists (e.g., not social coders)"
author: "*Compiled on `r date()` by `r Sys.info()['user']`*"
output:
html_document:
toc: true
number_sections: true
theme: cerulean
highlight: haddock
includes:
in_header: ohi_hdr.html
pdf_document:
toc: true
modified.png
staged.png

4.8.4 Push



4.9 Explore remote Github

The files you added should be on github.com:



Let's also explore commit history, file history.

4.9.1 Your turn!

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

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

4.10 Create a new R Markdown file

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

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

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

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

```
---
title: "My Project"
author: "Julie"
date: "11/21/2017"
output: html_document
---

# Data wrangling with dplyr

We are going use "gapminder" data to learn `dplyr`. It's going to be amazing.
```

Now, let's save it. I'm going to call my file `wrangle-dplyr.Rmd`.

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

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

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

Now, let's write some R code.

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

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

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

4.10.1 Your turn

Add a few more commands to your file from this morning. Execute them by trying the three ways above.

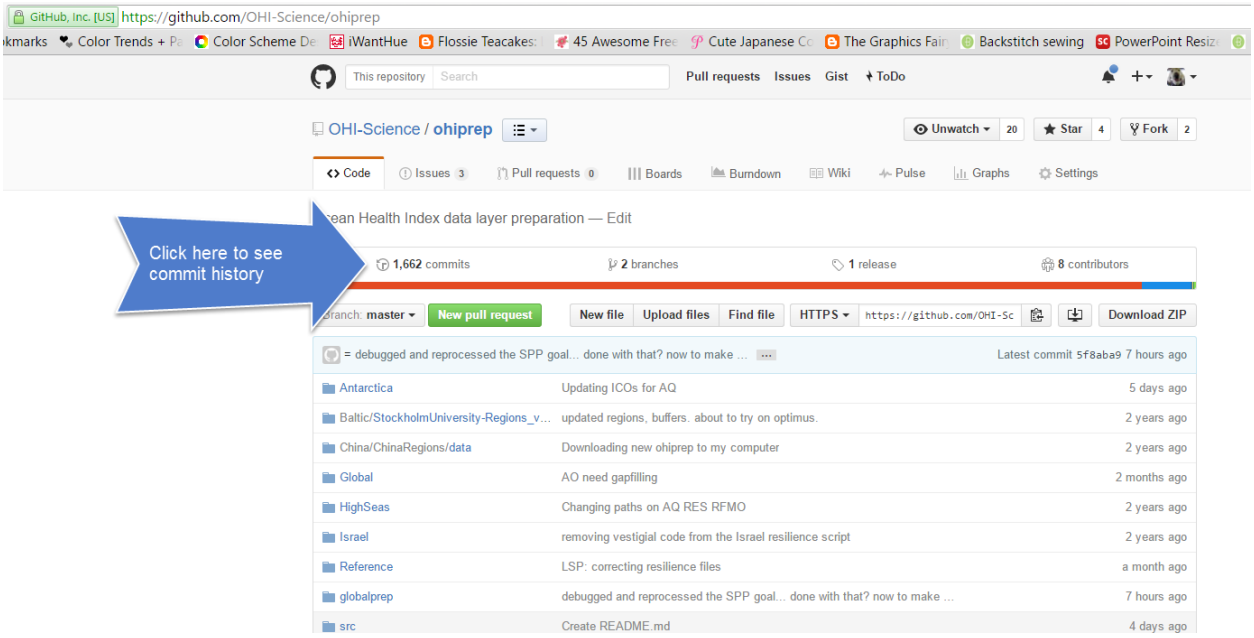
Then, sync your file to GitHub.

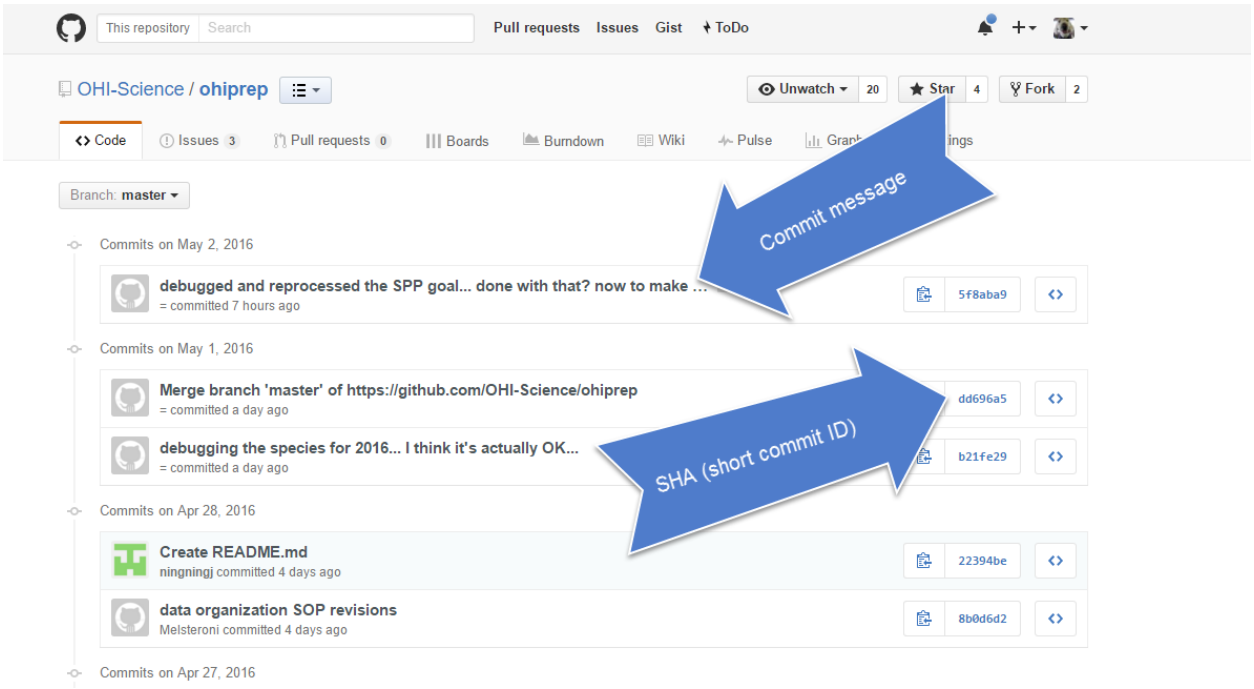
4.11 Committing - how often? Tracking changes in your files

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

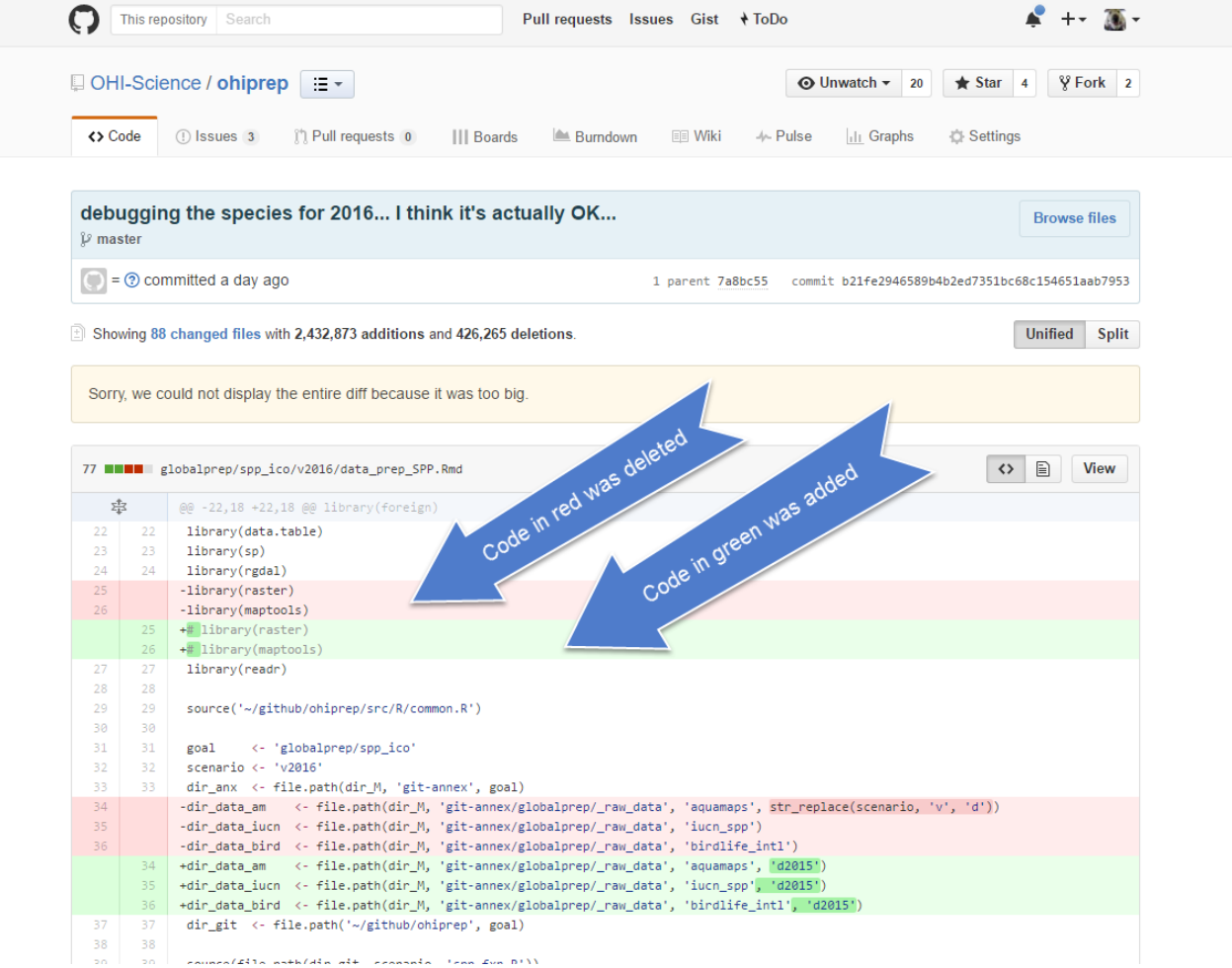
I tend to do this every time I finish a task (basically when I start getting nervous that I will lose my work). Once something is committed, it is very difficult to lose it.

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





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



4.12 Troubleshooting

If you have problems, we'll help you out using Jenny Bryan's HappyGitWithR, particularly the sections on Detect Git from RStudio and RStudio, Git, GitHub Hell (troubleshooting).

Chapter 5

Visualizing: `ggplot2`

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

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

This lesson borrows heavily from Hadley Wickham's R for Data Science book, and the Data Carpentry R for Ecology curriculum.

5.1 Objectives & Resources

5.1.1 Objectives

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

5.1.2 Resources

Here are some additional resources for data visualization in R:
- ggplot2-cheatsheet-2.0.pdf
- Interactive Plots and Maps - Environmental Informatics
- Graphs with ggplot2 - Cookbook for R
- ggplot2 Essentials - STHDA
- "Why I use ggplot2" - David Robinson Blog Post

5.2 Install our first package: `tidyverse`

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

So far we've been using packages that are already included in *base R*. These can be considered *out-of-the-box* packages and include things such as `sum` and `mean`. You can also download and install packages created by

the vast and growing R user community. The most traditional place to download packages is from CRAN, the Comprehensive R Archive Network. This is where you went to download R originally, and will go again to look for updates. You can also install packages directly from GitHub, which we'll do tomorrow.

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

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

```
## from CRAN:
install.packages("tidyverse") ## do this once only to install the package on your computer.
```

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

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

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

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

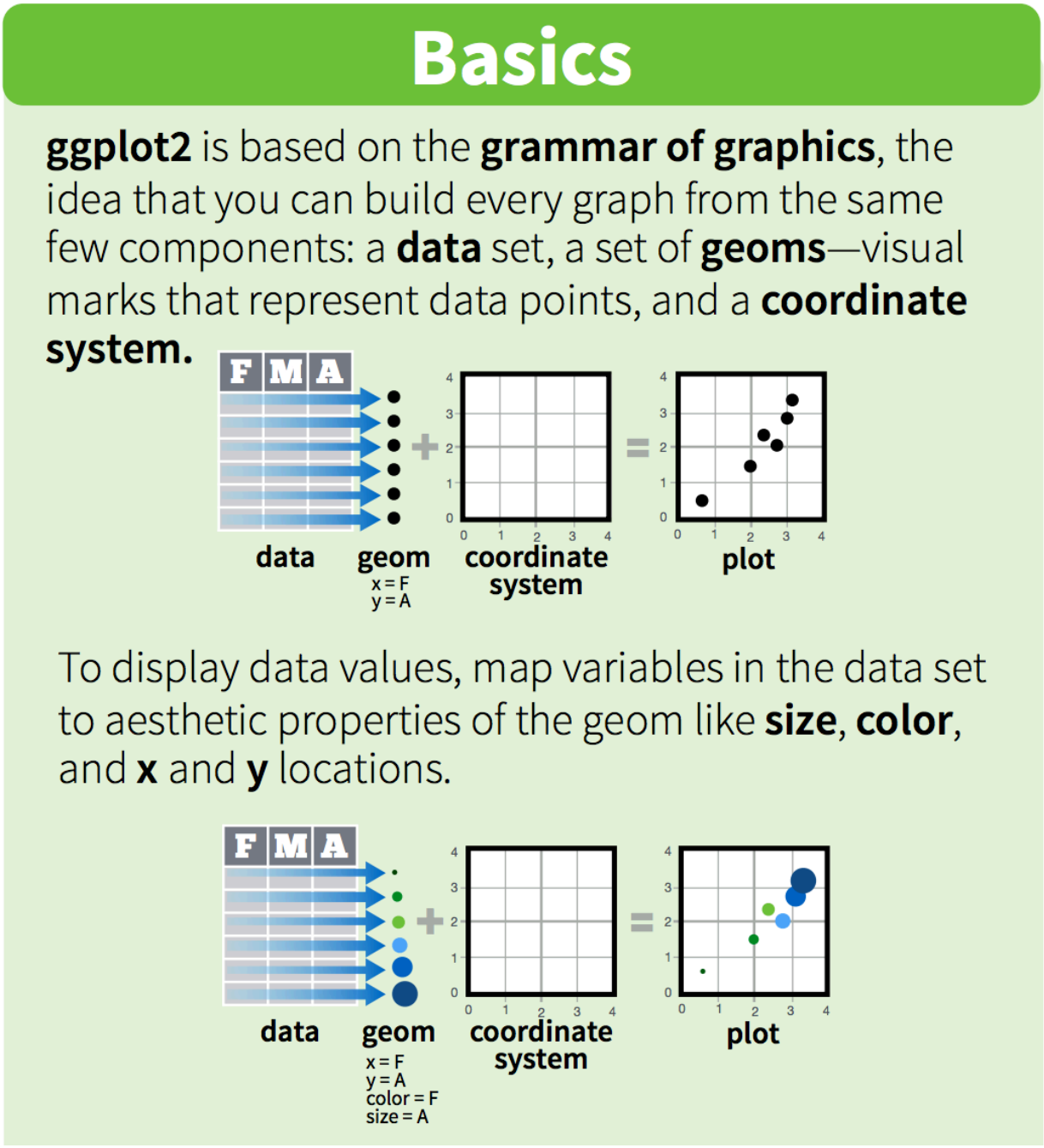
You can also install packages by going to the Packages tab in the bottom right pane. You can see the packages that you have installed (listed) and loaded (checkbox). You can also install packages using the install button, or check to see if any of your installed packages have updates available (update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

5.3 Plotting with ggplot2

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

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

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



5.4 Data

We are going to use the `mpg` dataset which provides information on fuel economy data for 38 car models. This data comes preloaded with the `tidyverse` so it is already loaded into R. Let's take a look at it.

```
mpg
```

```
## # A tibble: 234 x 11
##    manufacturer      model displ  year    cyl      trans    drv    cty    hwy
##           <chr>      <chr> <dbl> <int> <int>      <chr>  <chr> <int> <int>
```

```
##  1         audi         a4   1.8  1999     4   auto(l5)     f    18    29
##  2         audi         a4   1.8  1999     4 manual(m5)     f    21    29
##  3         audi         a4   2.0  2008     4 manual(m6)     f    20    31
##  4         audi         a4   2.0  2008     4   auto(av)     f    21    30
##  5         audi         a4   2.8  1999     6   auto(l5)     f    16    26
##  6         audi         a4   2.8  1999     6 manual(m5)     f    18    26
##  7         audi         a4   3.1  2008     6   auto(av)     f    18    27
##  8         audi a4 quattro   1.8  1999     4 manual(m5)     4    18    26
##  9         audi a4 quattro   1.8  1999     4   auto(l5)     4    16    25
## 10         audi a4 quattro   2.0  2008     4 manual(m6)     4    20    28
## # ... with 224 more rows, and 2 more variables: fl <chr>, class <chr>
```

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

1. `displ`, a car's engine size, in litres.

2. `hwy`, a car's fuel efficiency on the highway, in miles per gallon (mpg). A car with a low fuel efficiency consumes more fuel than a car with a high fuel efficiency when they travel the same distance.

To learn more about `mpg`, open its help page by running `?mpg`.

Now we're going to visualize the data that is held in this dataframe.

To build a ggplot, we need to:

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

```
ggplot(data = mpg)
```

- define aesthetics (`aes`), by selecting the variables to be plotted and the variables to define the presentation such as plotting size, shape color, etc.

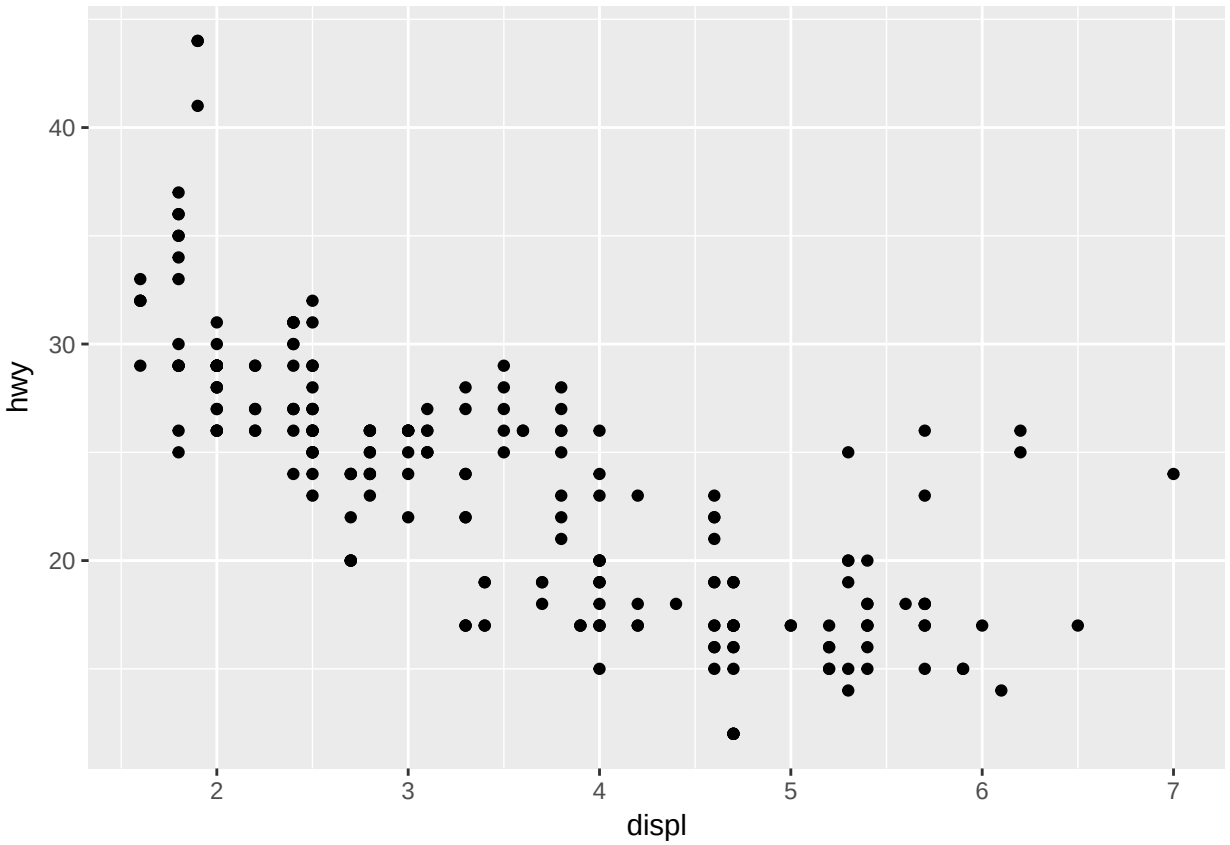
Run this code to put `displ` on the x-axis and `hwy` on the y-axis:

```
ggplot(data = mpg, aes(x = displ, y = hwy))
```

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

To add a geom to the plot use `+` operator. Because we have two continuous variables, let's use `geom_point()` first:

```
ggplot(data = mpg, aes(x = displ, y = hwy)) +
  geom_point()
```



The + in the **ggplot2** package is particularly useful because it allows you to modify existing `ggplot` objects. This means you can easily set up plot "templates" and conveniently explore different types of plots, so the above plot can also be generated with code like this:

```
# Assign plot to a variable
car_plot <- ggplot(data = mpg, aes(x = displ, y = hwy))

# Draw the plot
car_plot +
    geom_point()
```

Notes:

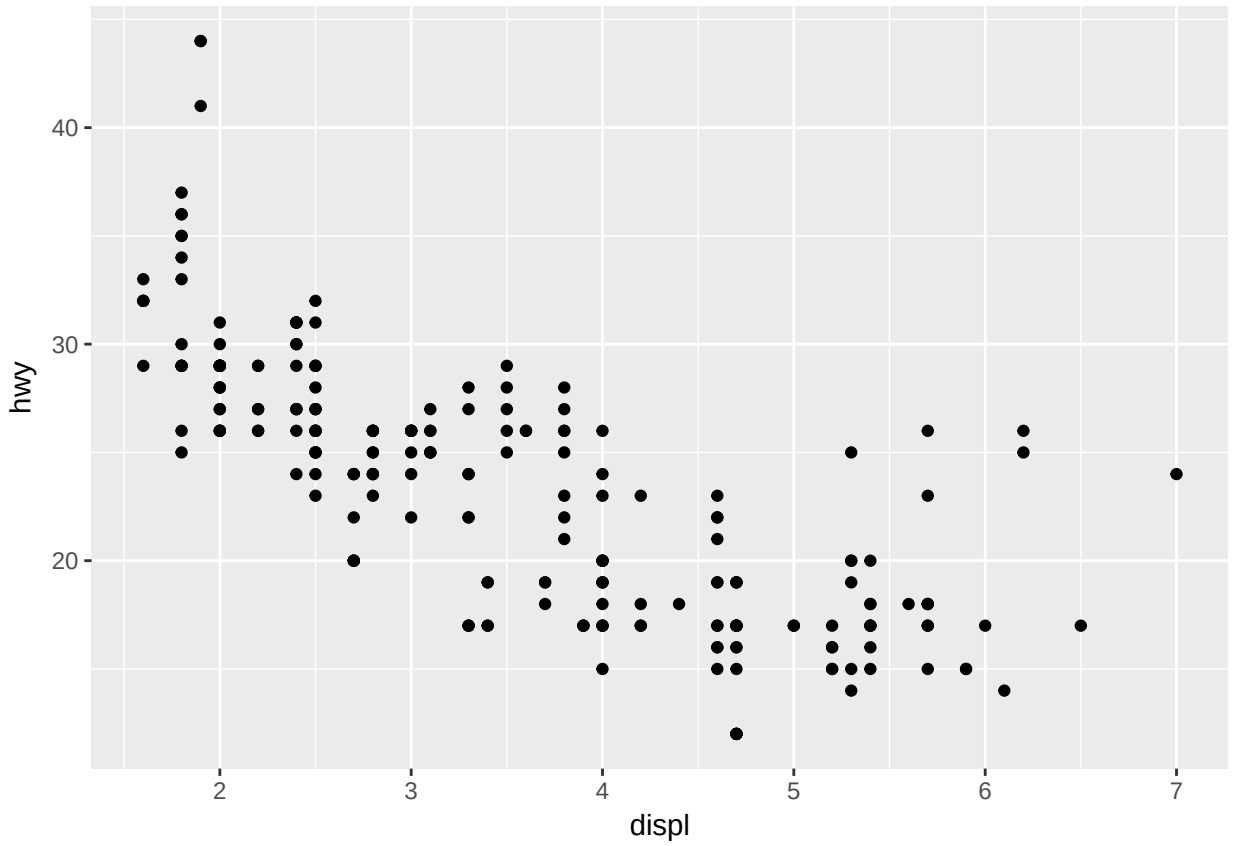
- Anything you put in the `ggplot()` function can be seen by any geom layers that you add (i.e., these are universal plot settings). This includes the x and y axis you set up in `aes()`.
- You can also specify aesthetics for a given geom independently of the aesthetics defined globally in the `ggplot()`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

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

5.5 Building your plots iteratively

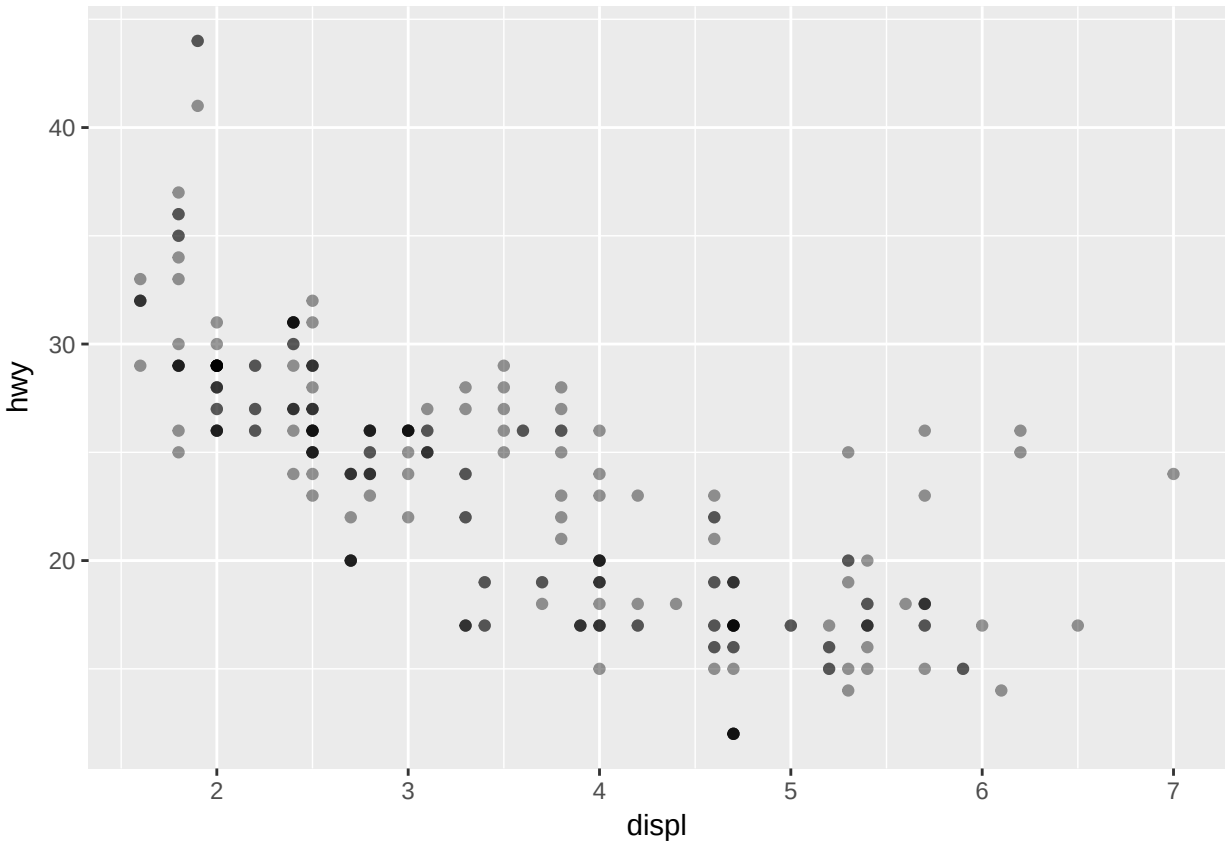
Building plots with ggplot is typically an iterative process. We start by defining the dataset we'll use, lay the axes, and choose a geom:

```
ggplot(data = mpg, aes(x = displ, y = hwy)) +
    geom_point()
```



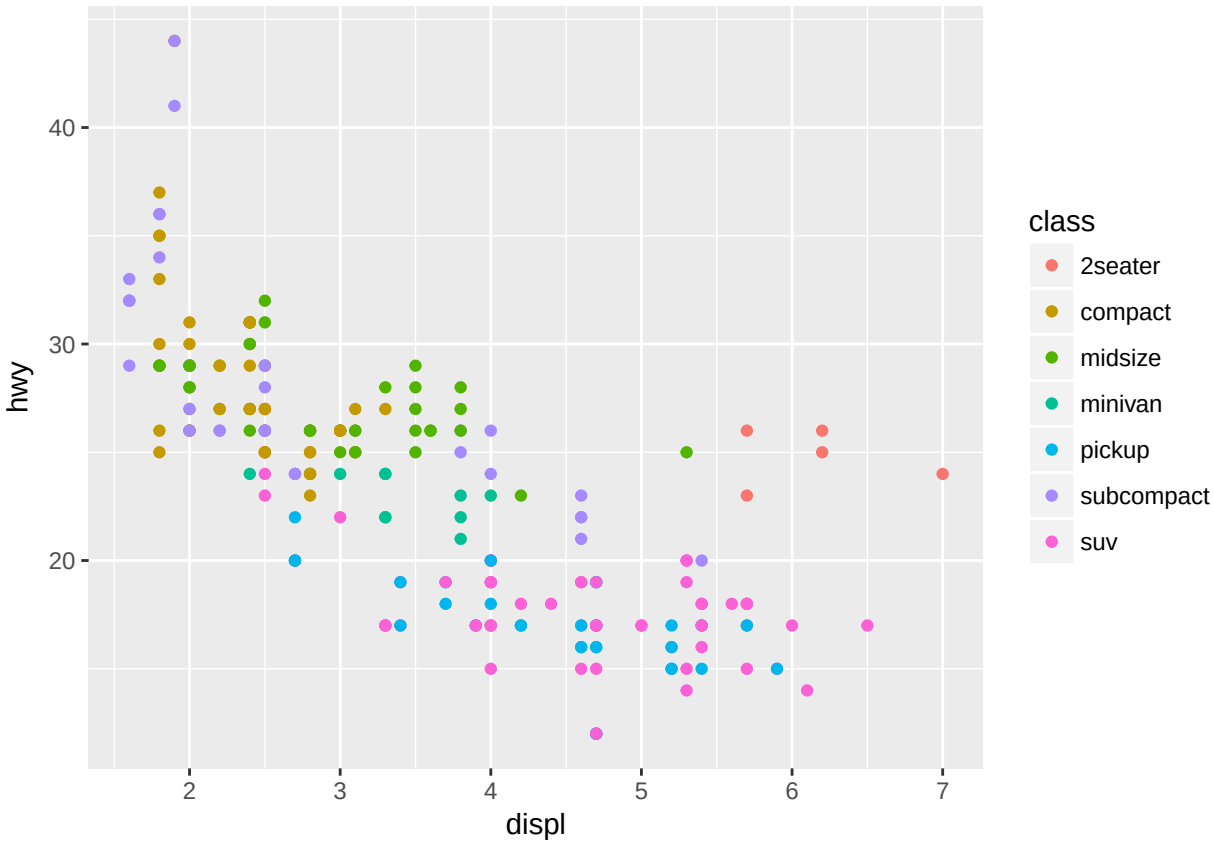
Then, we start modifying this plot to extract more information from it. For instance, we can add transparency (`alpha`) to avoid overplotting:

```
ggplot(data = mpg, aes(x = displ, y = hwy)) +
    geom_point(alpha = 0.4)
```



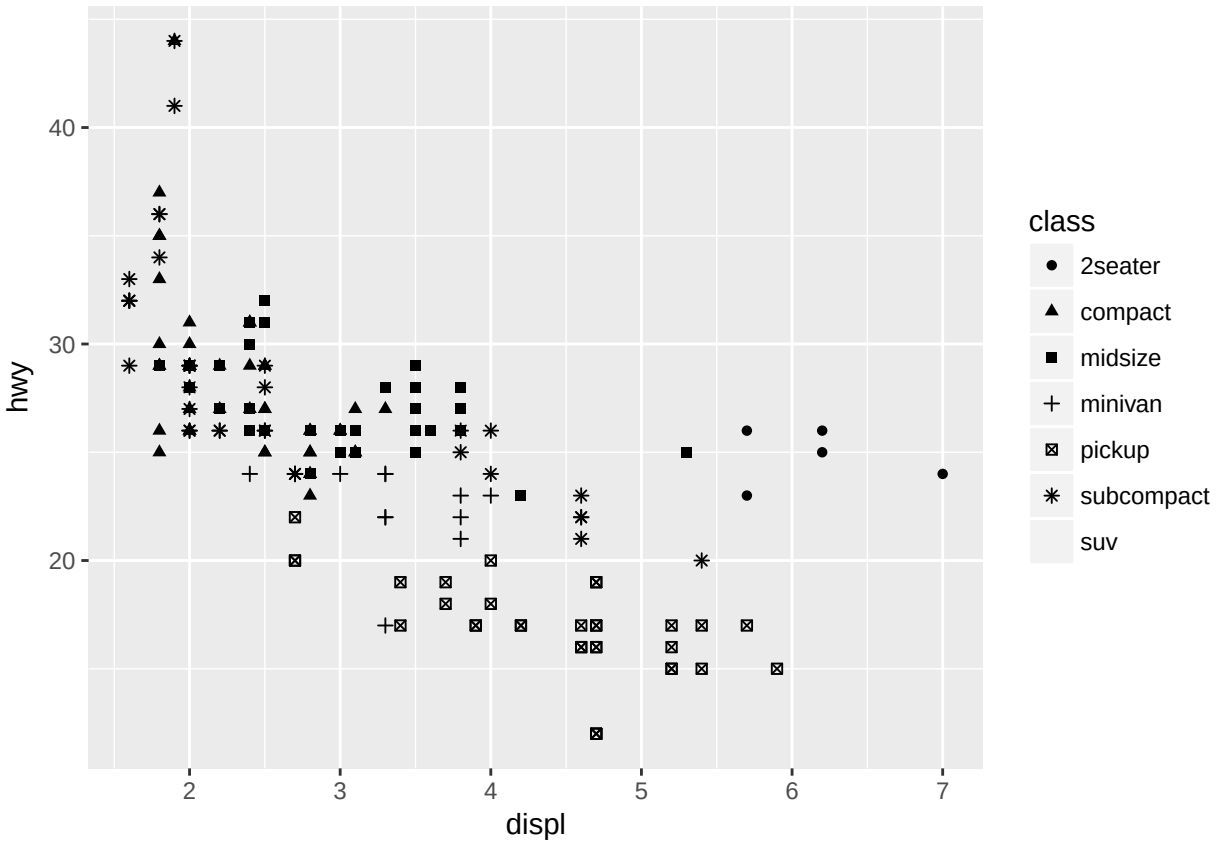
Or to color each species in the plot differently:

```
ggplot(data = mpg, aes(x = displ, y = hwy)) +
    geom_point(aes(color = class))
```



In the above example, we mapped `class` to the color aesthetic, but we could have mapped `class` to the shape aesthetic in the same way. In this case, the shape of each point would reveal its class affiliation.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, shape = class))
```



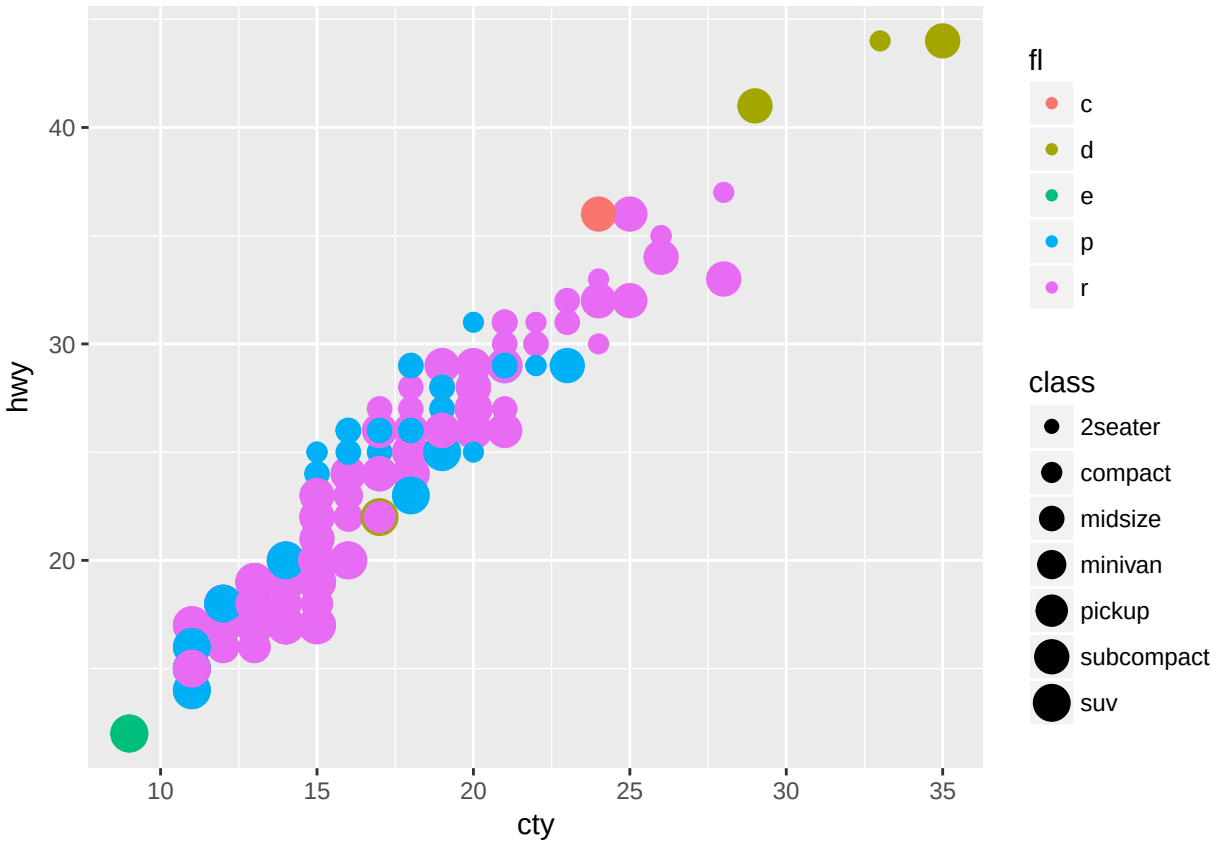
What happened to the SUVs? ggplot2 will only use six shapes at a time. By default, additional groups will go unplotted when you use the shape aesthetic.

5.5.1 Exercise

Make a scatterplot of `hwy` vs `cty` with different size points representing each car class and different colors for each fuel type.

We get a *warning* here, because mapping an unordered variable (`class`) to an ordered aesthetic (`size`) is not a good idea.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = cty, y = hwy, size = class, color = fl))
```



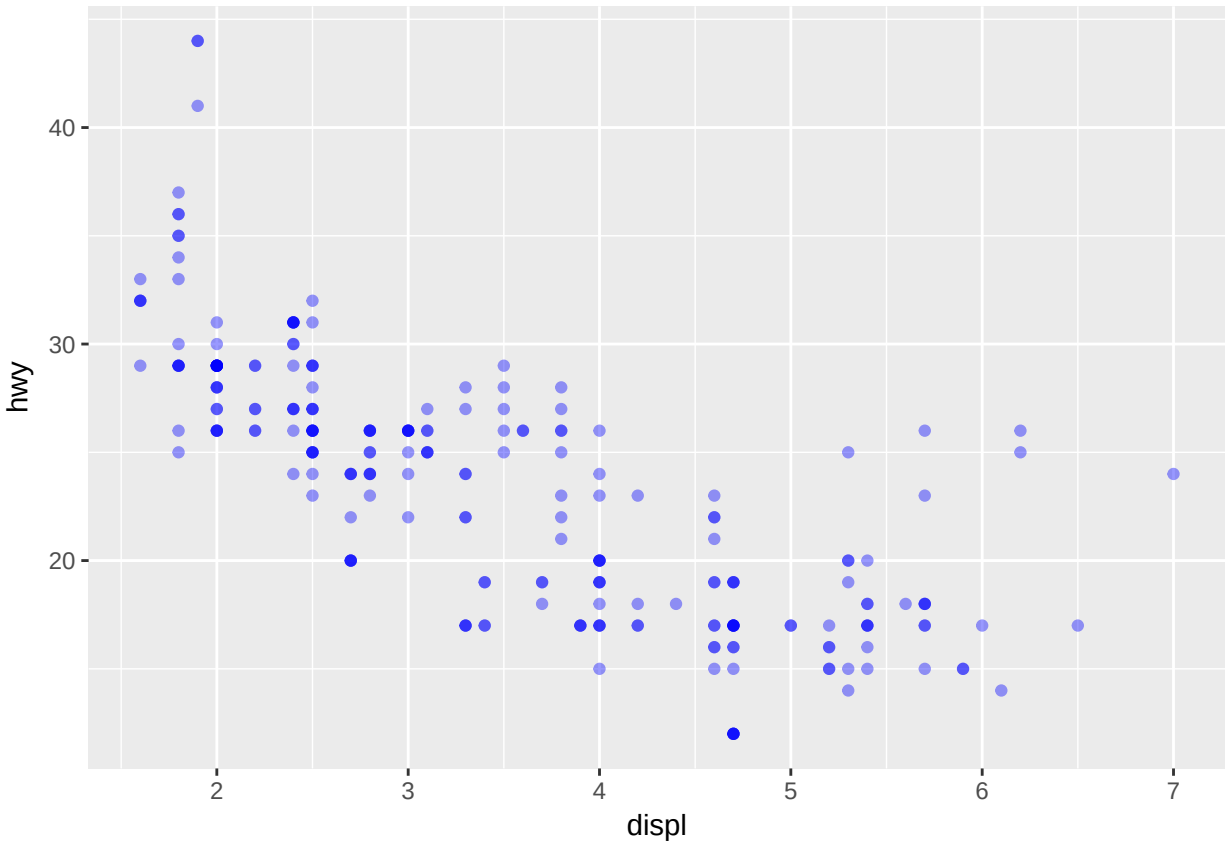
We can also add colors for all the points. Here, the color doesn't convey information about a variable, but only changes the appearance of the plot. To set an aesthetic manually, set the aesthetic by name as an argument of your geom function; i.e. it goes *outside* of `aes()`. You'll need to pick a value that makes sense for that aesthetic:

- The name of a color as a character string.

- The size of a point in mm.

- The shape of a point as a number.

```
ggplot(data = mpg, aes(x = displ, y = hwy)) +
    geom_point(alpha = 0.4, color = "blue")
```



5.5.2 Exercise

1. What's gone wrong with this code?

```
ggplot(data = mpg) +
  geom_point(aes(x = displ, y = hwy, color = "blue"))
```

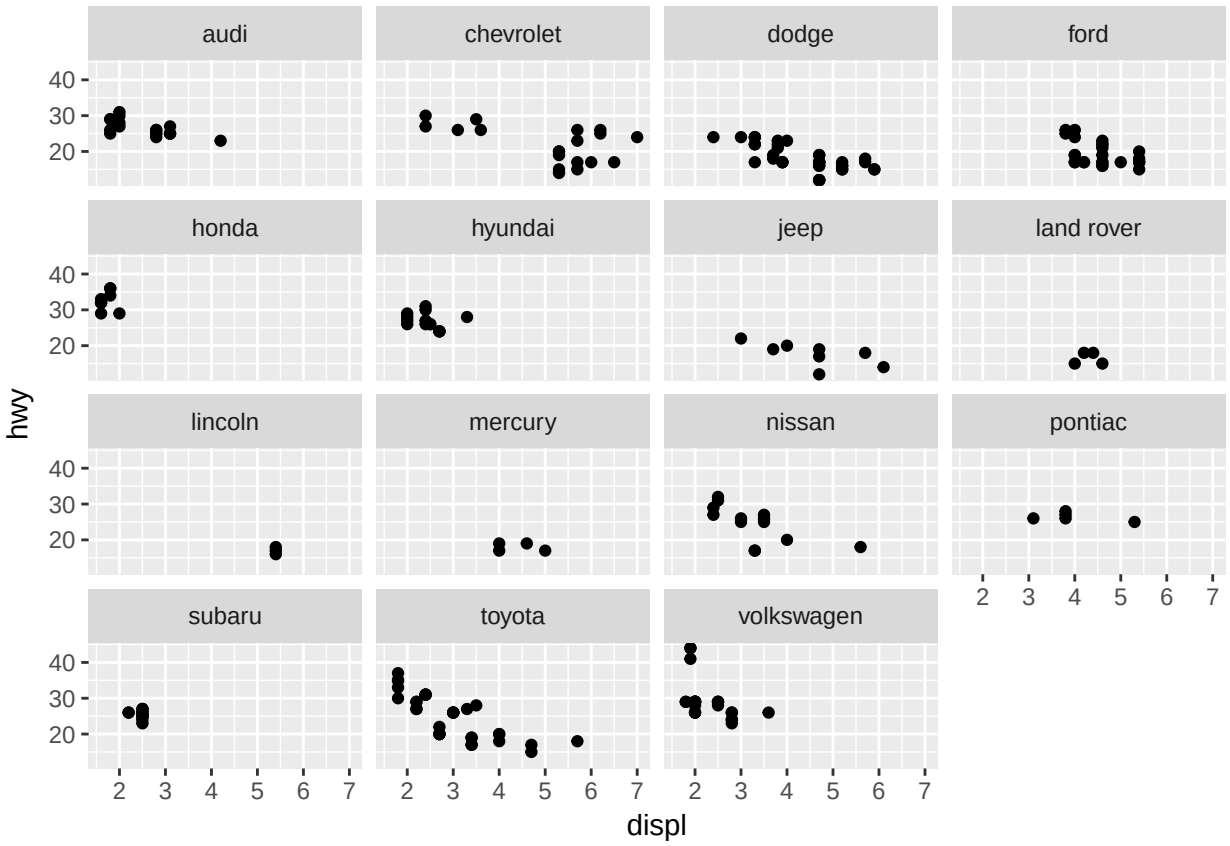
2. Plot `hwy` vs `displ` and have the point color to indicate `cty` mpg.
3. Now instead of color, use shape to indicate `cty` mpg. Why are these two aesthetics behaving differently?
4. What happens if you map an aesthetic to something other than a variable name, like `aes(colour = displ < 5)`?

STOP: commit, pull and push to github

5.6 Faceting

ggplot has a special technique called *faceting* that allows the user to split one plot into multiple plots based on a factor included in the dataset. We will use it to make a time series plot for each car manufacturer:

```
ggplot(data = mpg, aes(x = displ, y = hwy)) +
    geom_point() +
    facet_wrap(~ manufacturer)
```

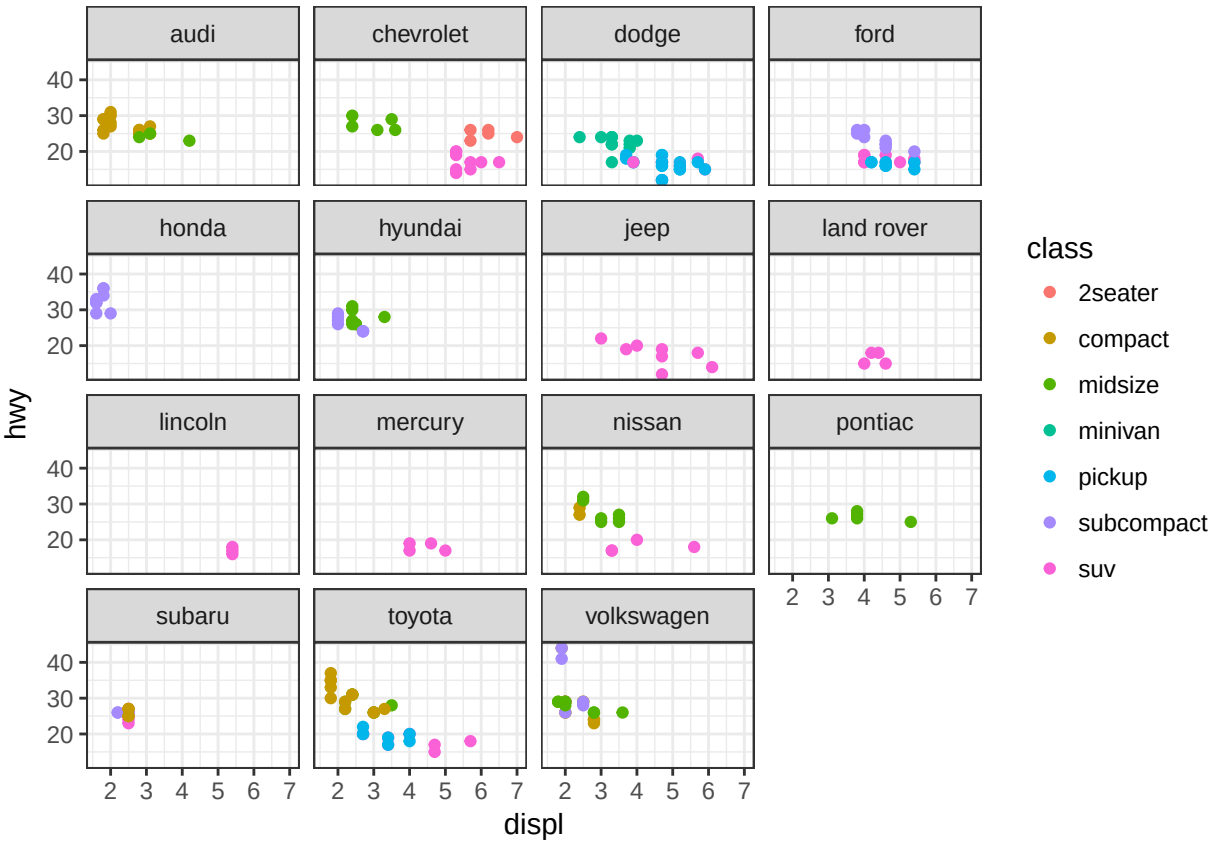


We can now make the faceted plot by splitting further by class using `color` (within a single plot):

```
ggplot(data = mpg, aes(x = displ, y = hwy, color = class)) +
    geom_point() +
    facet_wrap(~ manufacturer)
```

Usually plots with white background look more readable when printed. We can set the background to white using the function `theme_bw()`.

```
ggplot(data = mpg, aes(x = displ, y = hwy, color = class)) +
    geom_point() +
    facet_wrap(~ manufacturer) +
    theme_bw()
```



5.7 ggplot2 themes

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

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

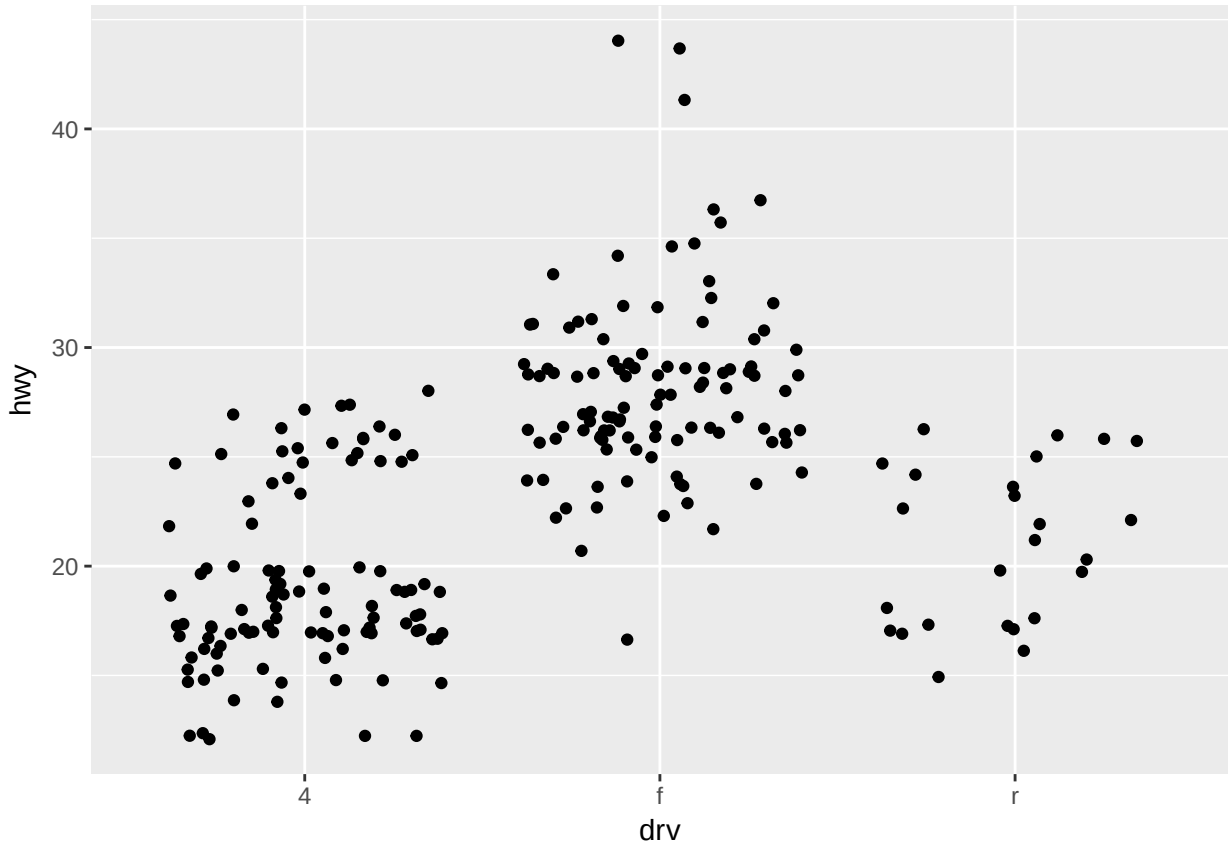
5.7.1 Exercise

Spend a couple minutes trying out your plot with different plot themes. The complete list of themes in **ggplot2** is available at http://docs.ggplot2.org/current/ggtheme.html. But for some more interesting themes, try installing the **ggthemes** package and using one of those themes. You can find more information on this package at https://cran.r-project.org/web/packages/ggthemes/vignettes/ggthemes.html

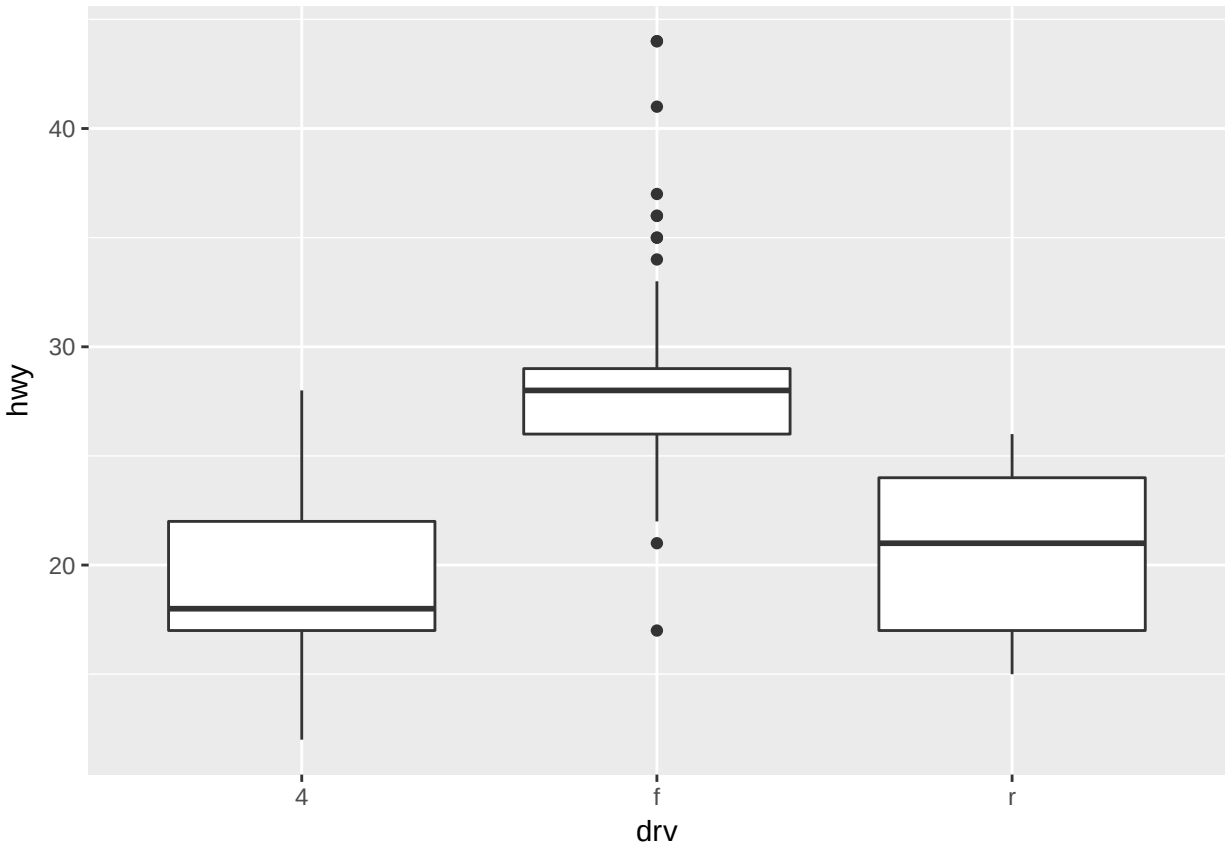
5.8 Geometric objects (geoms)

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

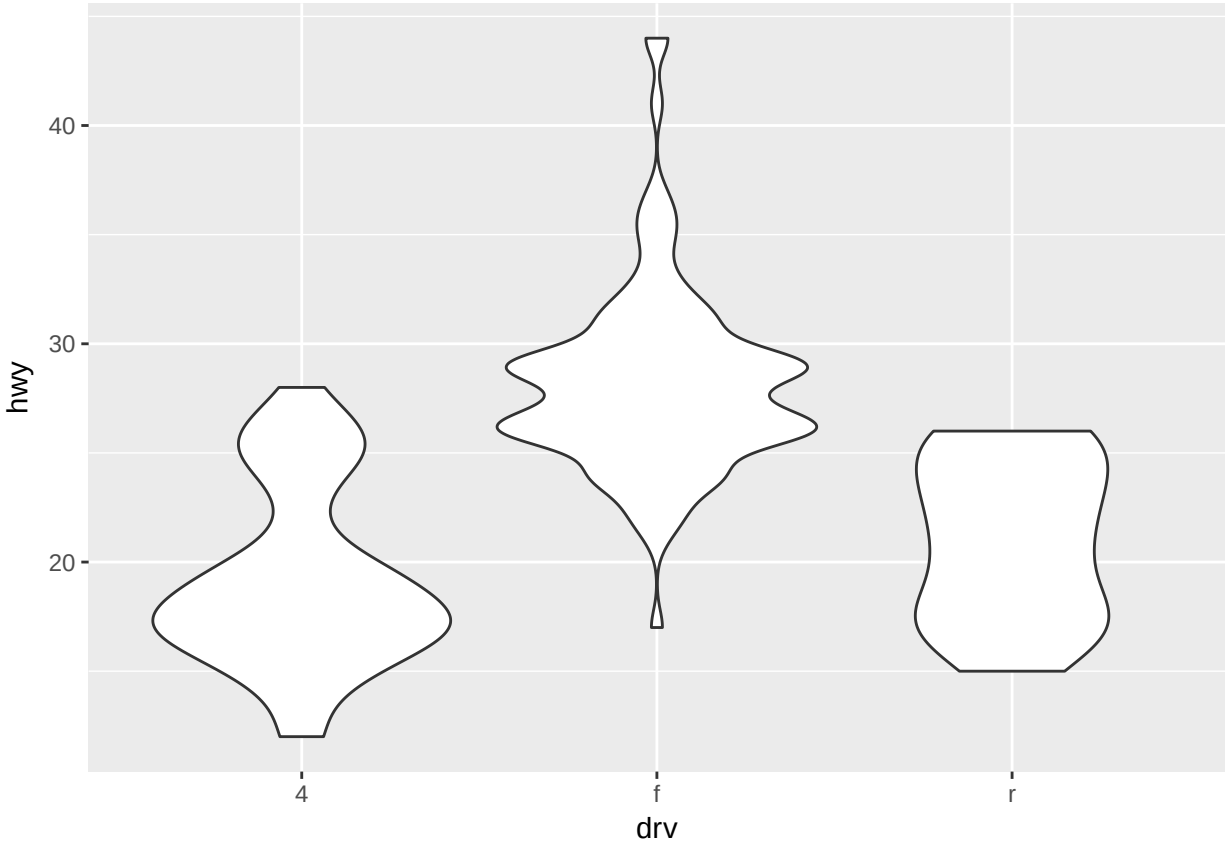
```
ggplot(mpg, aes(x = drv, y = hwy)) +
  geom_jitter()
```




```
ggplot(mpg, aes(x = drv, y = hwy)) +
  geom_boxplot()
```

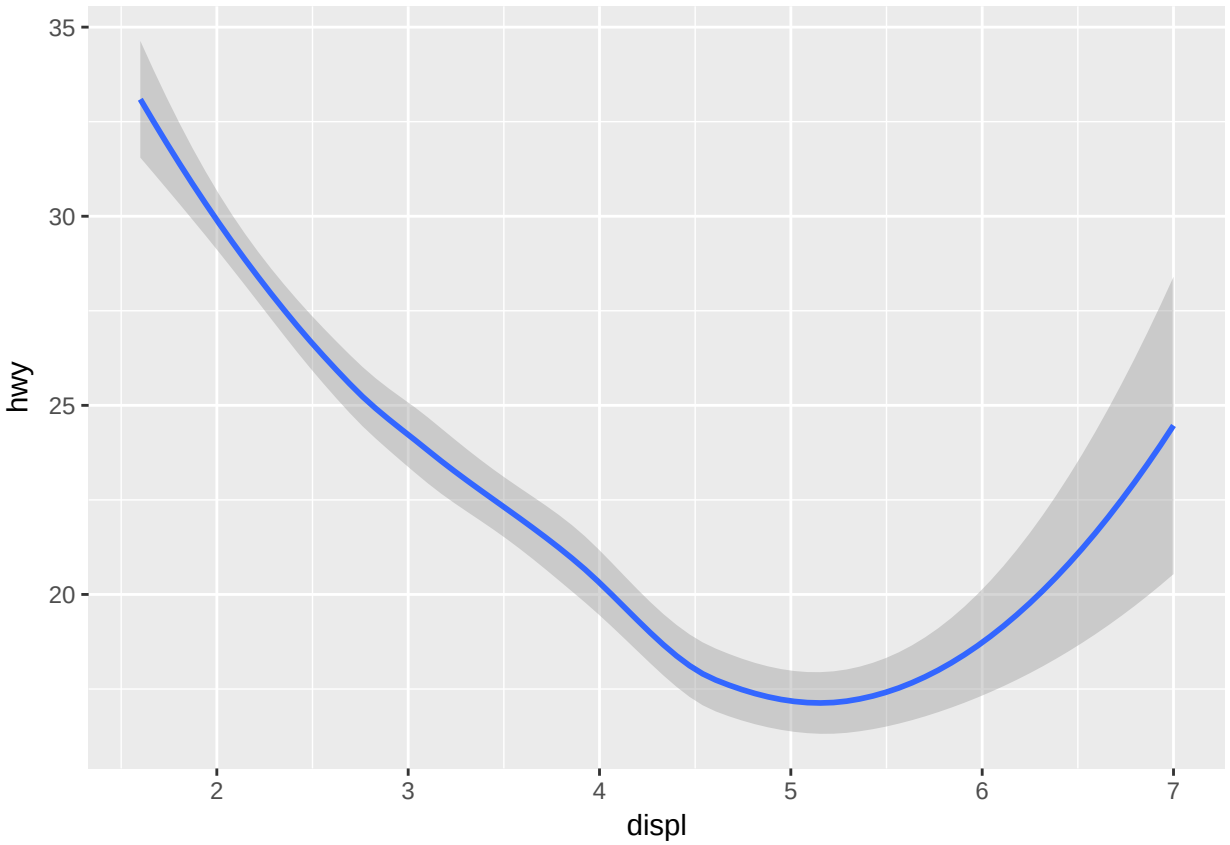



```
ggplot(mpg, aes(x = drv, y = hwy)) +
  geom_violin()
```



To plot a smoothed mean from the data above, use `geom_smooth`.

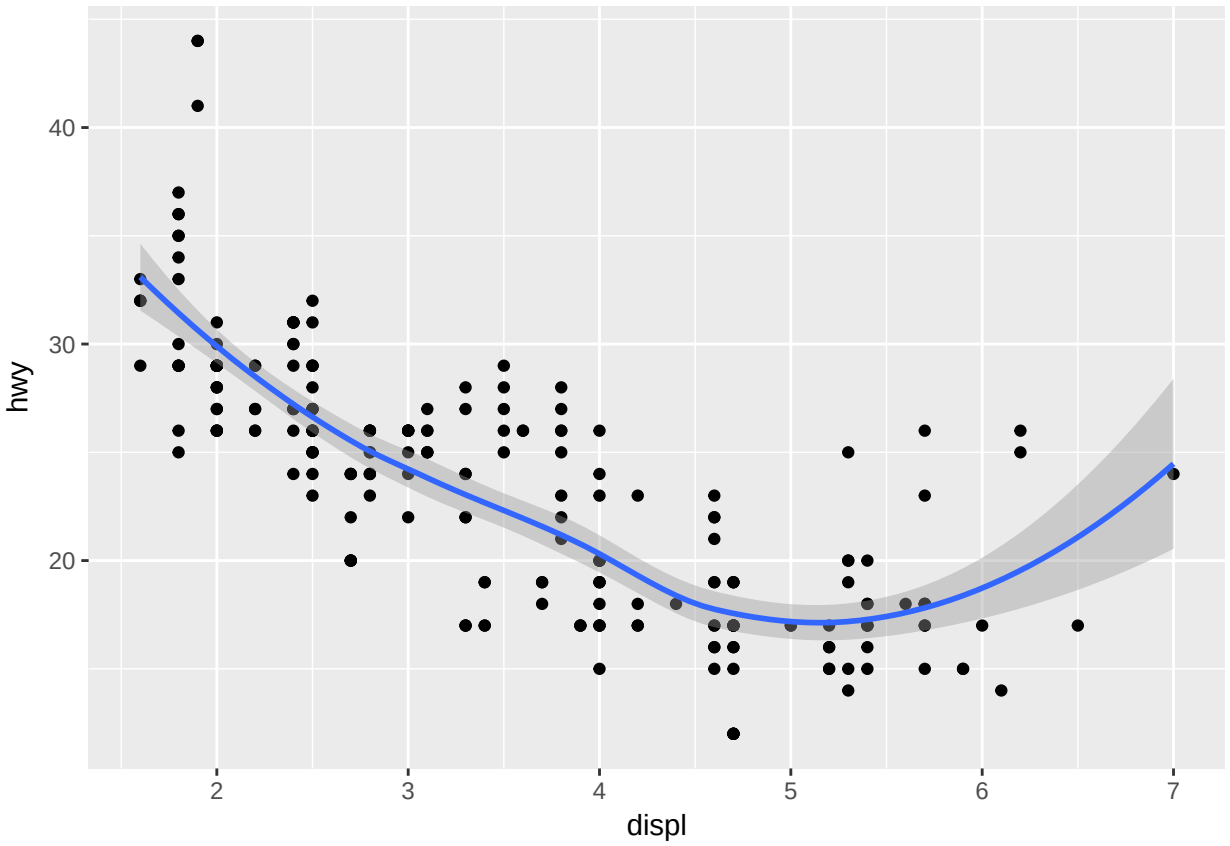
```
ggplot(data = mpg) +
  geom_smooth(aes(x = displ, y = hwy))
```



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

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

```
ggplot(data = mpg) +
  geom_point(aes(x = displ, y = hwy)) +
  geom_smooth(aes(x = displ, y = hwy))
```



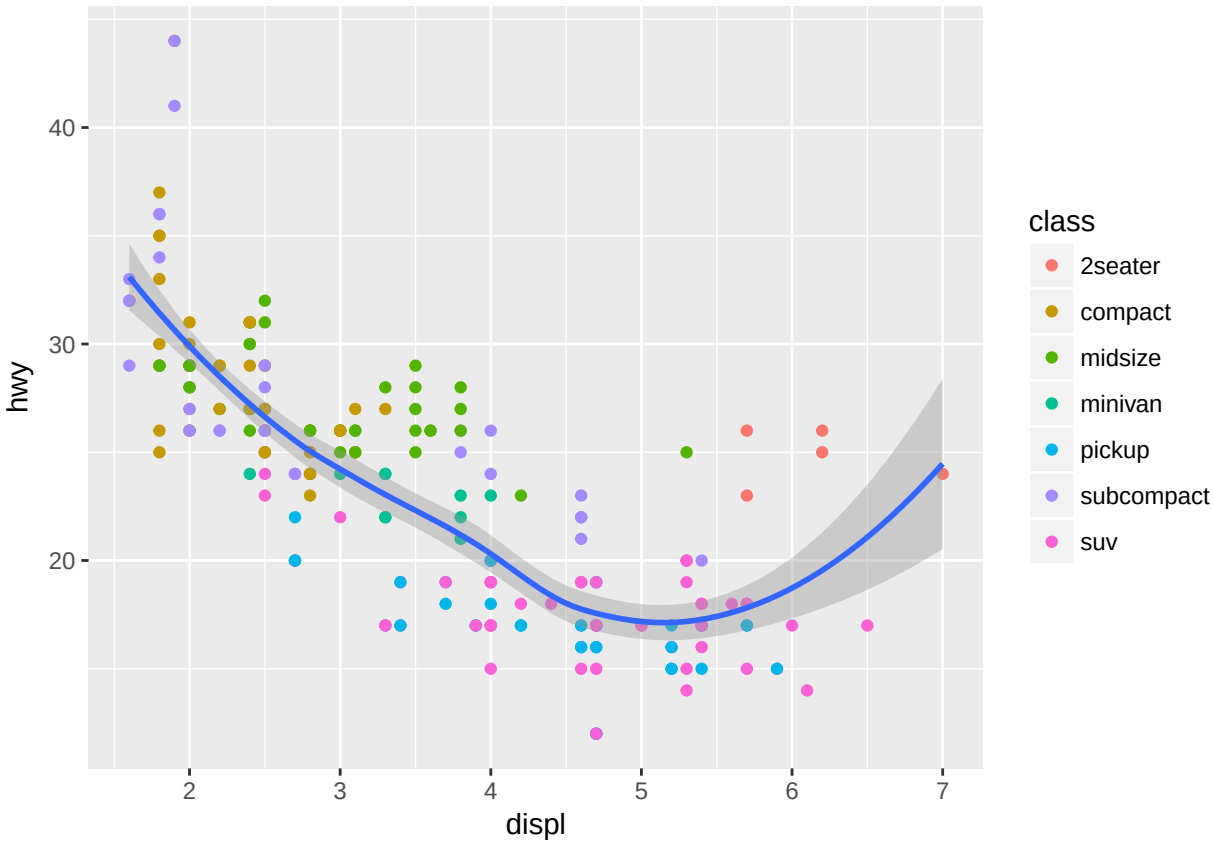
Notice that this plot contains two geoms in the same graph!

This, however, introduces some duplication in our code. Imagine if you wanted to change the y-axis to display `cty` instead of `hwy`. You'd need to change the variable in two places, and you might forget to update one. You can avoid this type of repetition by passing a set of mappings to `ggplot()`. ggplot2 will treat these mappings as global mappings that apply to each geom in the graph. In other words, this code will produce the same plot as the previous code:

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth()
```

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

```
ggplot(data = mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth()
```

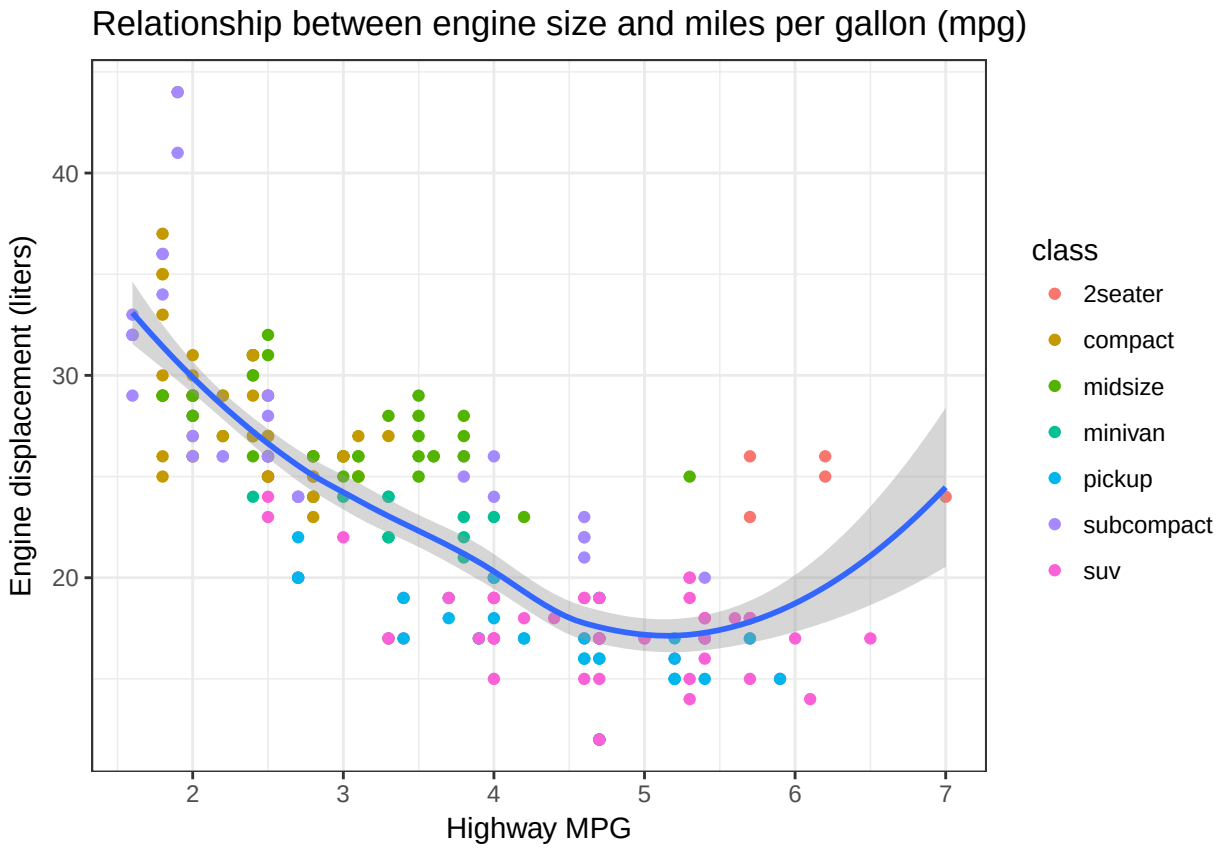


5.9 Customization

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

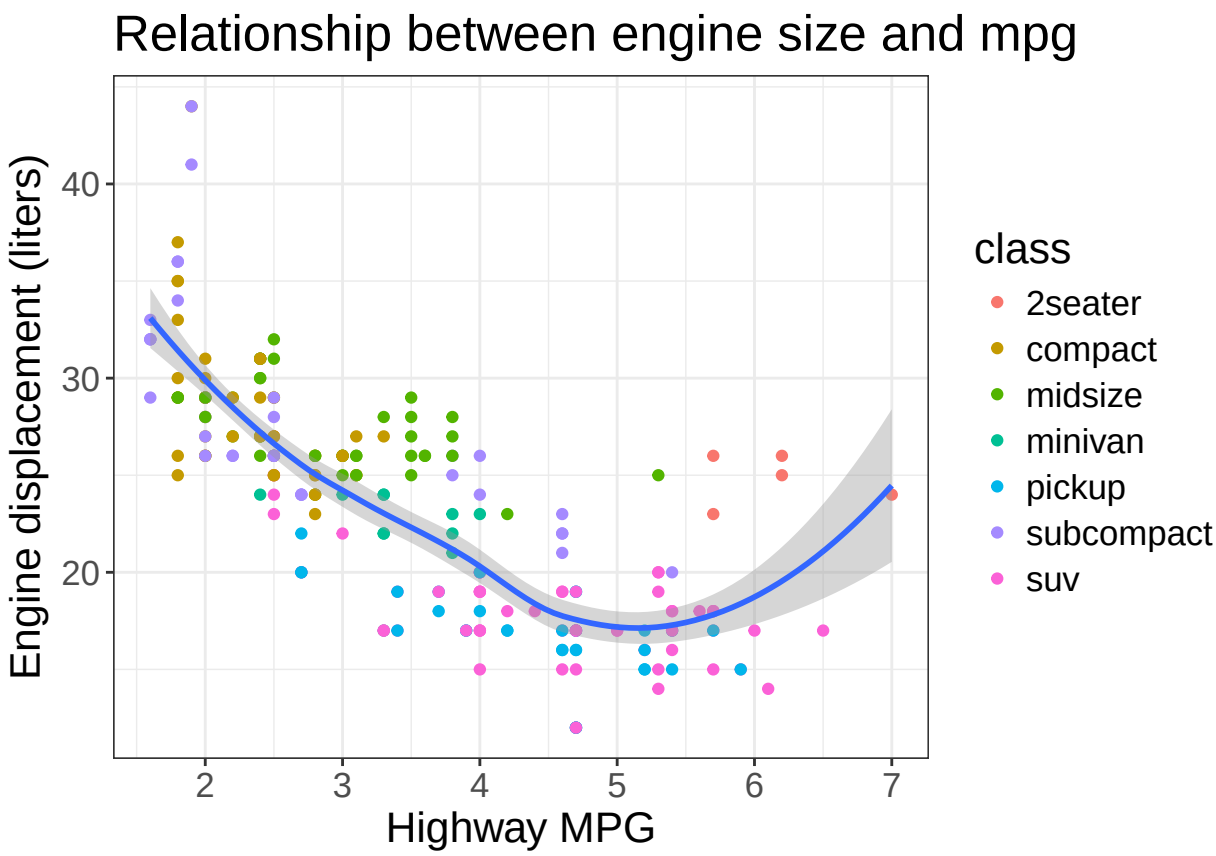
Now, let's change names of axes to something more informative than 'hwy' and 'displ' and add a title to the figure:

```
ggplot(data = mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth() +
  labs(title = "Relationship between engine size and miles per gallon (mpg)",
         x = "Highway MPG",
         y = "Engine displacement (liters)") +
  theme_bw()
```



The axes have more informative names, but their readability can be improved by increasing the font size:

```
ggplot(data = mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth() +
  labs(title = "Relationship between engine size and mpg",
         x = "Highway MPG",
         y = "Engine displacement (liters)") +
  theme_bw() +
  theme(text=element_text(size = 16))
```



5.9.1 Challenge

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

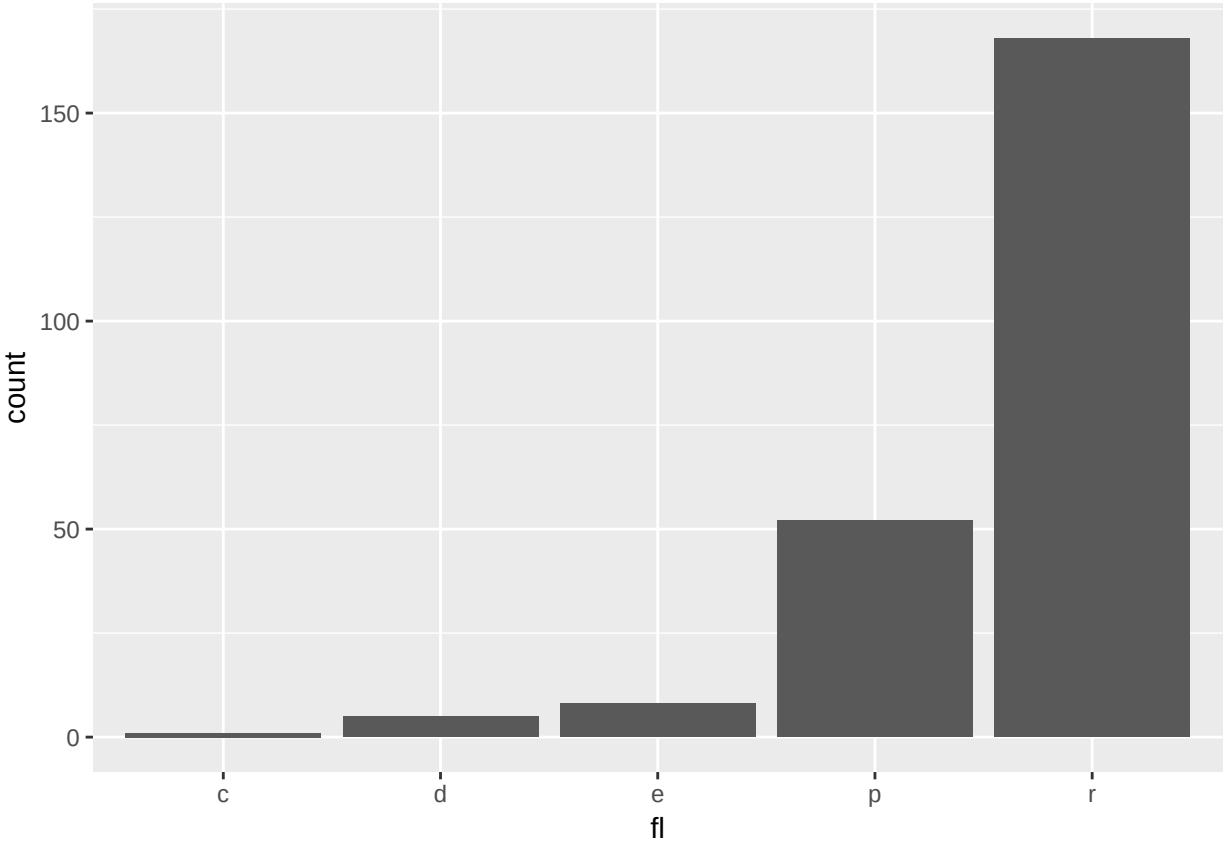
Here are some ideas:

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

5.10 Bar charts

Next, let's take a look at a bar chart. Bar charts seem simple, but they are interesting because they reveal something subtle about plots. Consider a basic bar chart, as drawn with `geom_bar()`. The following chart displays the total number of cars in the `mpg` dataset, grouped by `fl` (fuel type).

```
ggplot(data = mpg) +
  geom_bar(aes(x = fl))
```



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

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

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

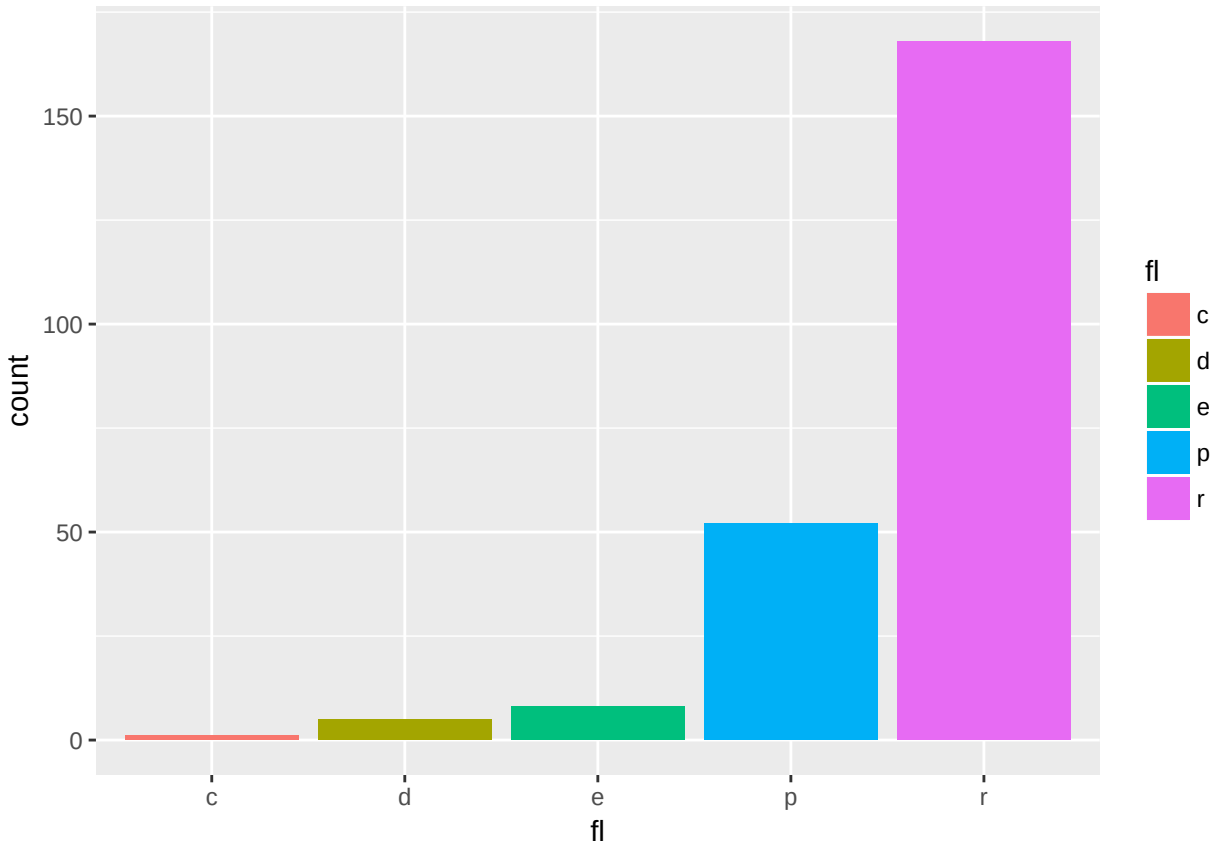
You can learn which stat a geom uses by inspecting the default value for the `stat` argument. For example, `?geom_bar` shows that the default value for `stat` is "count", which means that `geom_bar()` uses `stat_count()`. `stat_count()` is documented on the same page as `geom_bar()`, and if you scroll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

ggplot2 provides over 20 stats for you to use. Each stat is a function, so you can get help in the usual way, e.g. `?stat_bin`. To see a complete list of stats, try the ggplot2 cheatsheet.

5.10.1 Position adjustments

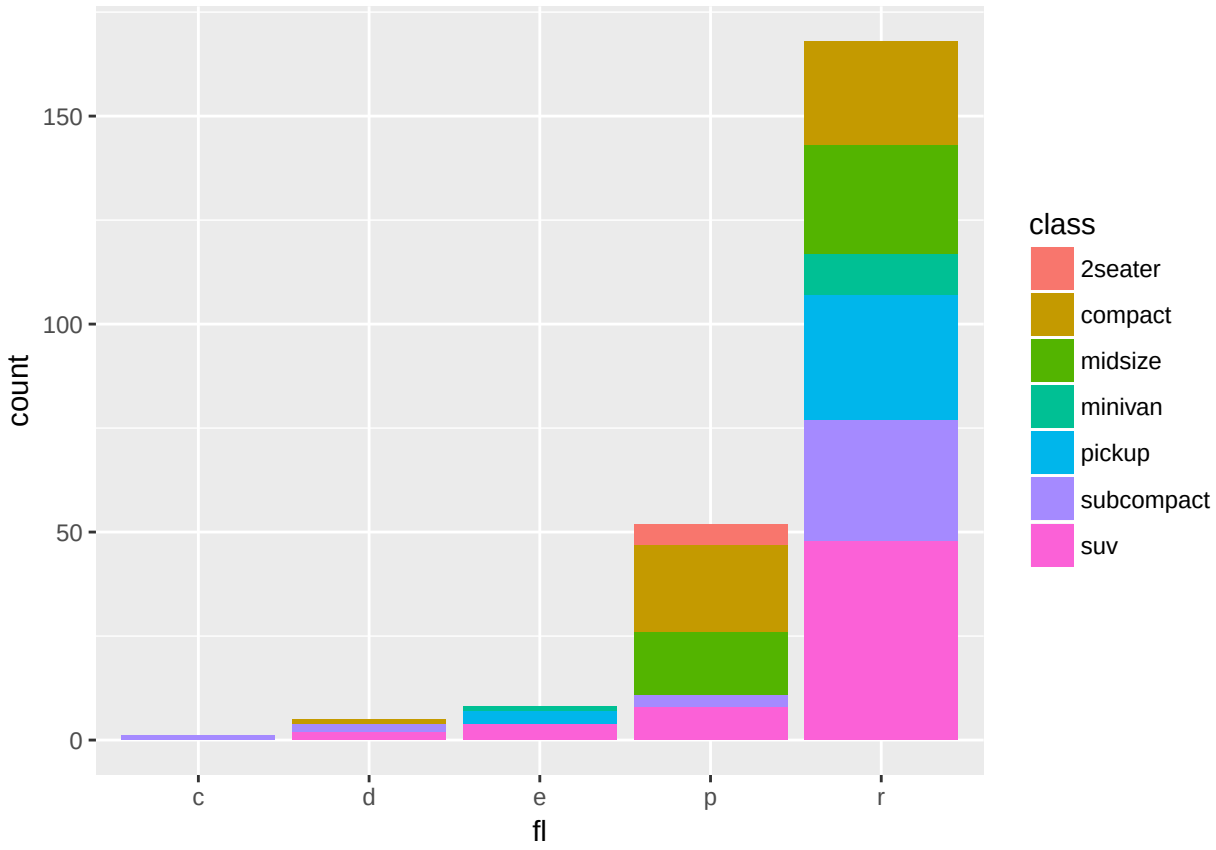
There's one more piece of magic associated with bar charts. You can colour a bar chart using either the `color` aesthetic, or, more usefully, `fill`:

```
ggplot(data = mpg) +
  geom_bar(aes(x = fl, fill = fl))
```



This isn't particularly useful since both the geom and the color are displaying the same information. Instead, map the fill aesthetic to another variable, like `class`: the bars are automatically stacked. Each colored rectangle represents a combination of `fl` and `class`.

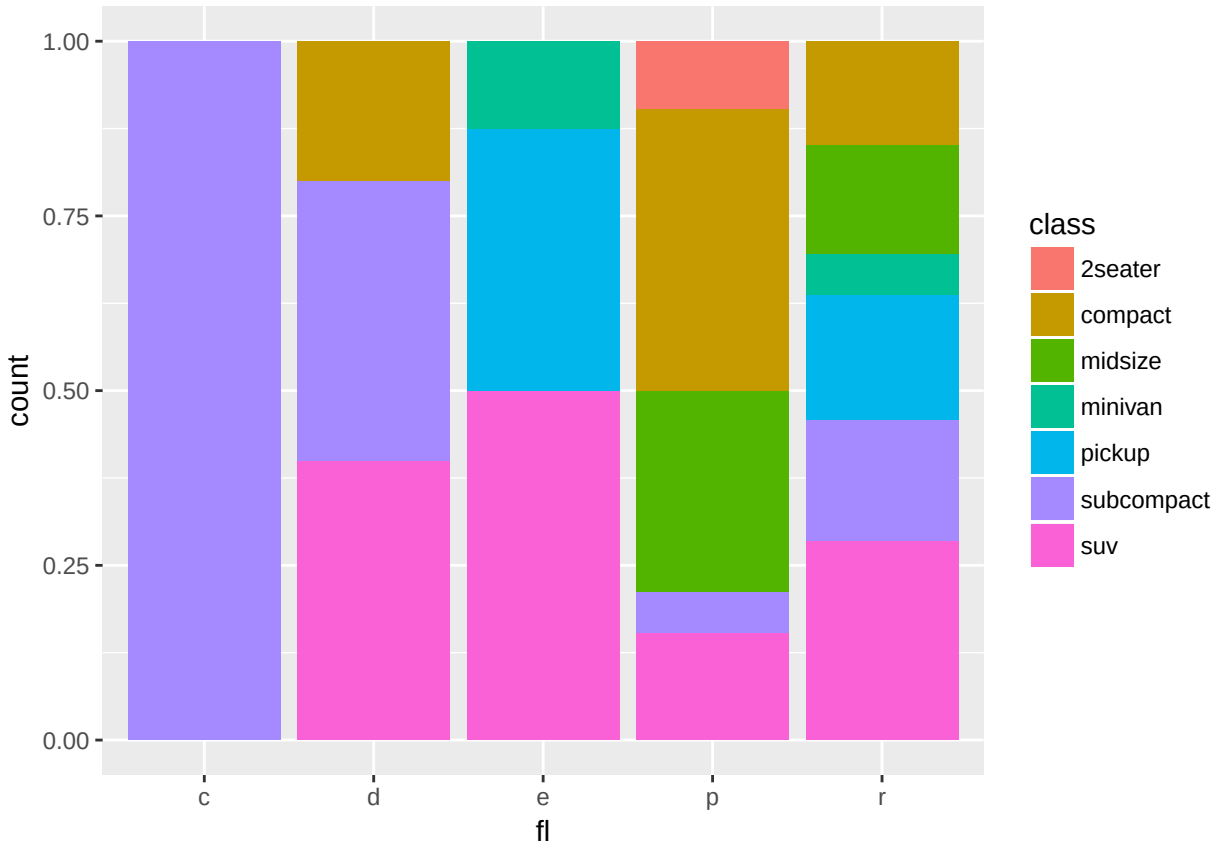
```
ggplot(data = mpg) +
  geom_bar(aes(x = fl, fill = class))
```



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

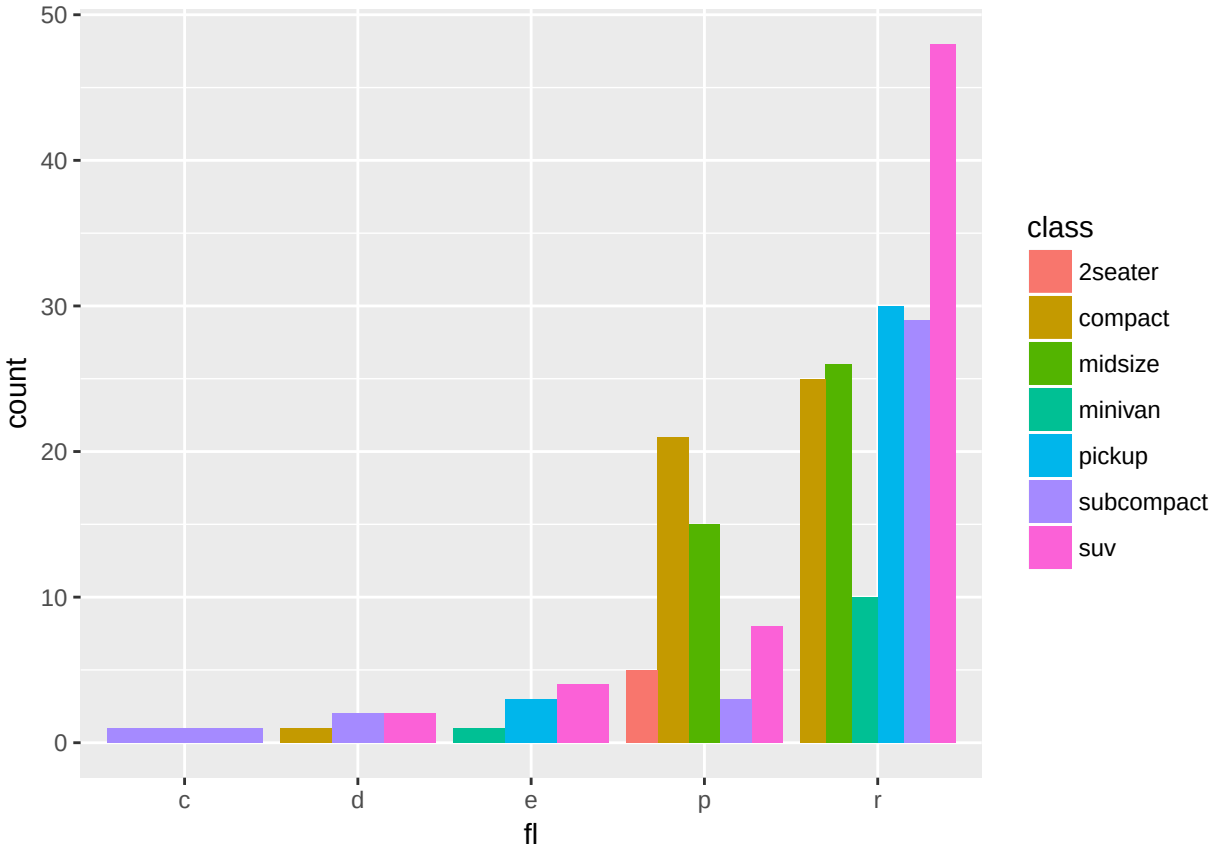
- `position = "fill"` works like stacking, but makes each set of stacked bars the same height. This makes it easier to compare proportions across groups.

```
ggplot(data = mpg) +
  geom_bar(aes(x = fl, fill = class), position = "fill")
```



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

```
ggplot(data = mpg) +
  geom_bar(aes(x = fl, fill = class), position = "dodge")
```



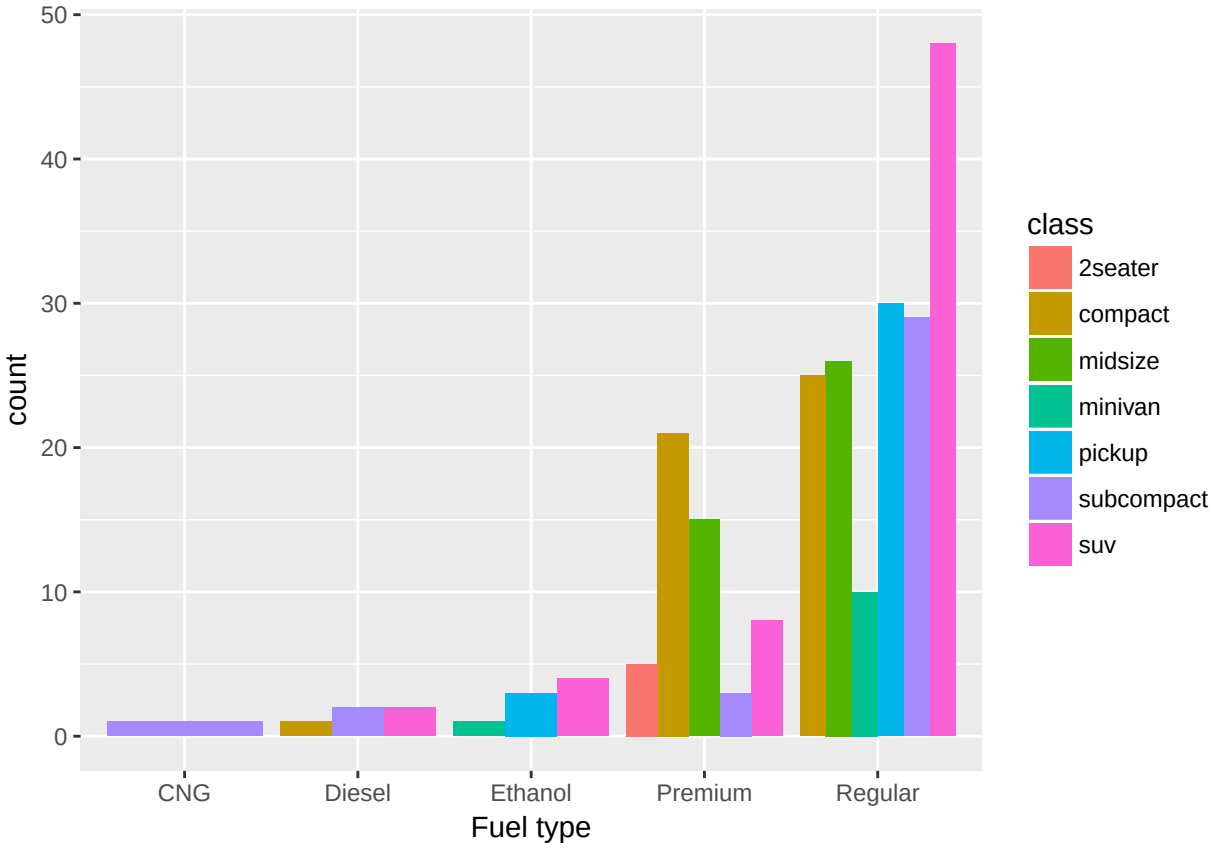
5.10.2 Challenge

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

Here are some ideas:

- Plot different variables using `geom_bar()`. For what variables is this geom useful?
- Flip the x and y axes.
- Use `scale_x_discrete` to change the x-axis tick labels to "CNG", "Diesel", "Ethanol", "Premium", and "Regular"

```
ggplot(data = mpg) +
      geom_bar(aes(x = fl, fill = class), position = "dodge") +
   scale_x_discrete(labels=c("CNG", "Diesel", "Ethanol", "Premium", "Regular")) +
  xlab("Fuel type")
```



5.11 Arranging and exporting plots

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

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

```
my_plot <- ggplot(data = mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth() +
  labs(title = "Relationship between engine size and mpg",
       x = "Highway MPG",
       y = "Engine displacement (liters)") +
  theme_bw() +
  theme(text=element_text(size = 16))

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

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

5.12 Save and push to GitHub

Chapter 6

Data Wrangling: `dplyr`

> Data scientists, according to interviews and expert estimates, spend from 50 percent to 80 percent of their time mired in the mundane labor of collecting and preparing data, before it can be explored for useful information. - NYTimes (2014)

6.1 Overview of `dplyr`

We are going to introduce you to data wrangling in R first with the tidyverse. The tidyverse is a new suite of packages that match a philosophy of data science developed by Hadley Wickham and the RStudio team. I find it to be a more straight-forward way to learn R. We will also show you by comparison what code will look like in "Base R", which means, in R without any additional packages (like the "tidyverse" package) installed. I like David Robinson's blog post on the topic of teaching the tidyverse first.

For some things, base-R is more straight forward, and we'll show you that too. Whenever we use a function that is from the tidyverse, we will prefix it so you'll know for sure.

Objectives

- learn about tidy data
- learn `dplyr` with `gapminder` data
- practice RStudio-GitHub workflow

Resources

Today's materials are again borrowing from some excellent sources, including:

- Jenny Bryan's lectures from STAT545 at UBC: Introduction to dplyr
- Hadley Wickham and Garrett Grolemund's R for Data Science
- Software Carpentry's R for reproducible scientific analysis materials: Dataframe manipulation with dplyr
- First developed for Software Carpentry at UCSB
- RStudio's data wrangling cheatsheet
- RStudio's data wrangling webinar

6.2 Prerequisites

R Skill Level: Beginner - you've got basics of R down and are ready to wrangle your data.

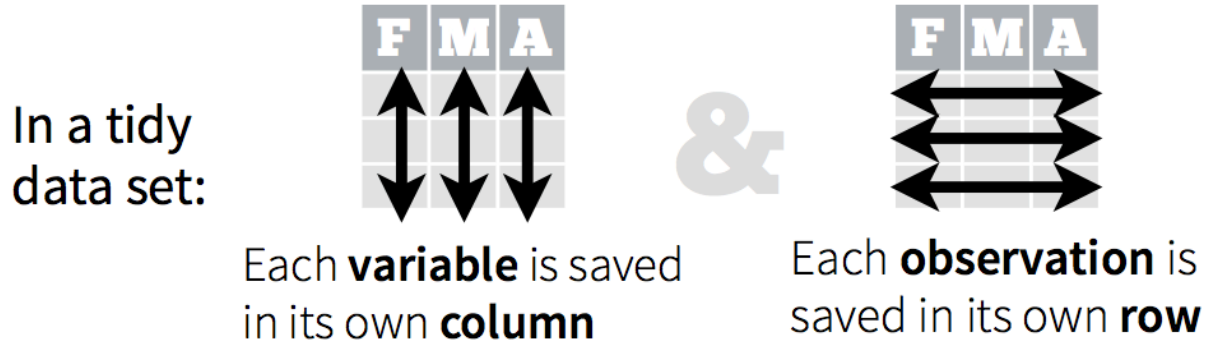
We will use the `dplyr` package, which will have been installed with:

```
install.packages('tidyverse')
```

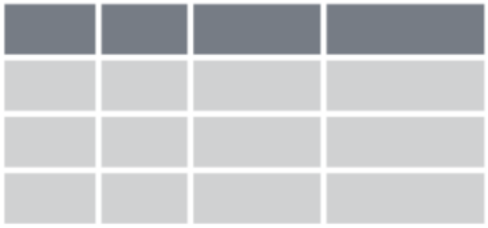
6.3 Tidy Data

Hadley Wickham, RStudio's Chief Scientist, has been building R packages for data wrangling and visualization based on the idea of **tidy data**.

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



The `mpg` dataset we were working with this morning was an example of tidy data. When data are tidy, you are



manufacturer	model	displ
audi	a4	1.
audi	a4	1.
audi	a4	2.
audi	a4	2.
audi	a4	2.
audi	a4	2.
audi	a4	3.
audi	a4 quattro	1.
audi	a4 quattro	1.
audi	a4 quattro	2.
audi	a4 quattro	2.

set up to work with it for your analyses, plots, etc.

Right now we are going to use `dplyr` to wrangle this tidyish data set (the transform part of the cycle), and then come back to tidying messy data using `tidyr` once we've had some fun wrangling. These are both part of the `tidyverse` package that we've already installed:

And actually, Hadley Wickham and RStudio have created a ton of packages that help you at every step of the way here. This is from one of Hadley's recent presentations:

6.3.1 Setup

We'll do this in a new RMarkdown file.

Here's what to do:

1. Clear your workspace (Session > Restart R)
2. New File > R Markdown…
3. Save as `gapminder-wrangle.Rmd`

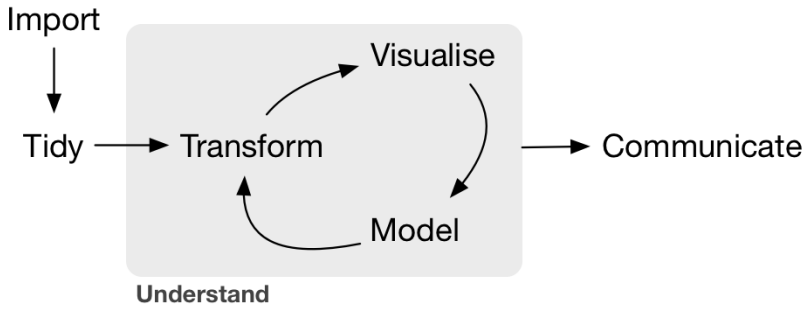


Figure 6.1:

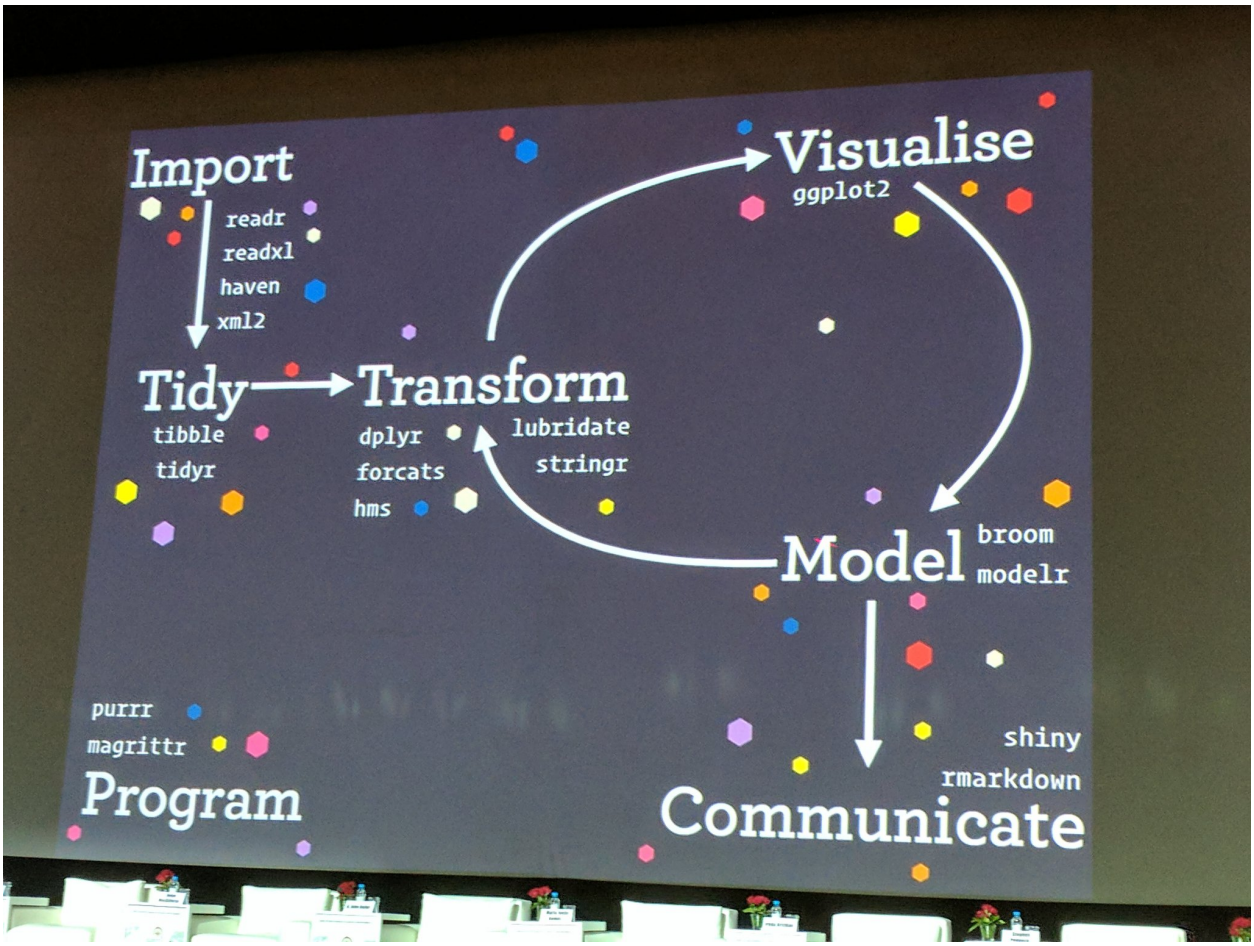


Figure 6.2:

country	year	pop	continent	lifeExp	gdpPercap
Afghanistan	1952	8425333	Asia	28.801	779.4453145
Afghanistan	1957	9240934	Asia	30.332	820.8530296
Afghanistan	1962	10267083	Asia	31.997	853.10071
Afghanistan	1967	11537966	Asia	34.02	836.1971382
Afghanistan	1972	13079460	Asia	36.088	739.9811058
Afghanistan	1977	14880372	Asia	38.438	786.11336
Afghanistan	1982	12881816	Asia	39.854	978.0114388
Afghanistan	1987	13867957	Asia	40.822	852.3959448
Afghanistan	1992	16317921	Asia	41.674	649.3413952
Afghanistan	1997	22227415	Asia	41.763	635.341351
Afghanistan	2002	25268405	Asia	42.129	726.7340548

Figure 6.3:

4. Delete the irrelevant text and write a little note to yourself about how we'll be wrangling gapminder data using dplyr. You can edit the title too if you need to.

6.3.2 load `tidyverse` (which has `dplyr` inside)

In your R Markdown file, let's make sure we've got our libraries loaded. Write the following:

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

This is becoming standard practice for how to load a library in a file, and if you get an error that the library doesn't exist, you can install the package easily by running the code within the comment (highlight `install.packages("tidyverse")` and run it).

6.4 Explore the gapminder data.frame

We will work with some of the data from the Gapminder project.

The data are on GitHub. Navigate there by going to:

github.com > ohi-science > data-science-training > data > gapminder.csv

or by copy-pasting this in the browser: `https://github.com/OHI-Science/data-science-training/blob/master/data/ga`

Have a look at the data. It's a .csv file, which you've probably encountered before, but GitHub has formatted it nicely so it's easy to look at. You can see that for every country and year, there are several columns with data in them.

6.4.1 read data with `readr::read_csv()`

We can read this data into R directly from GitHub, without downloading it. We can do that by clicking on the Raw button on the top-right of the data. This displays it as the raw csv file, without formatting. Copy the url:

https://raw.githubusercontent.com/jules32/2017-11-30-MBARI/gh-pages/data/gapminder.csv

Now, let's go back to RStudio. In our R Markdown, let's read this csv file and name the variable "gapminder". We will use the `read_csv()` function from the `readr` package (part of the tidyverse, so it's already installed!).

```
## read gapminder csv. Note the readr:: prefix identifies which package it's in
gapminder <- readr::read_csv('https://raw.githubusercontent.com/jules32/2017-11-30-MBARI/gh-pages/data/g
```

Let's inspect:

```
## explore the gapminder dataset
gapminder # this is super long! Let's inspect in different ways
```

Let's use `head` and `tail`:

```
head(gapminder) # shows first 6
tail(gapminder) # shows last 6

head(gapminder, 10) # shows first X that you indicate
tail(gapminder, 12) # guess what this does!
```

`str()` will provide a sensible description of almost anything: when in doubt, just `str()` some of the recently created objects to get some ideas about what to do next.

```
str(gapminder) # ?str - displays the structure of an object
```

`gapminder` is a `data.frame`. We aren't going to get into the other types of data receptacles today ('arrays', 'matrices'), because working with data.frames is what you should primarily use. Why?

- data.frames package related variables neatly together, great for analysis
- most functions, including the latest and greatest packages actually **require** that your data be in a data.frame
- data.frames can hold variables of different flavors such as
  - character data (country or continent names; "Characters (chr)")
  - quantitative data (years, population; "Integers (int)" or "Numeric (num)")
  - categorical information (male vs. female)

We can also see the `gapminder` variable in RStudio's Environment pane (top right)

More ways to learn basic info on a data.frame.

```
names(gapminder)
dim(gapminder)    # ?dim dimension
ncol(gapminder)   # ?ncol number of columns
nrow(gapminder)   # ?nrow number of rows
```

We can combine using `c()` to reverse-engineer `dim()`! Just a side-note here, but I wanted to introduce you to `c()`: we'll use it later.

```
c(nrow(gapminder), ncol(gapminder)) # ?c combines values into a vector or list.
```

A statistical overview can be obtained with `summary()`

```
summary(gapminder)
```

6.4.2 Look at the variables inside a data.frame

To specify a single variable from a data.frame, use the dollar sign `$`. The `$` operator is a way to extract of replace parts of an object–check out the help menu for `$`. It's a common operator you'll see in R.

```
gapminder$lifeExp # very long! hard to make sense of...
head(gapminder$lifeExp) # can do the same tests we tried before
```

```
str(gapminder$lifeExp) # it is a single numeric vector
summary(gapminder$lifeExp) # same information, just formatted slightly differently
```

6.5 dplyr basics

OK, so let's start wrangling with dplyr.

There are five `dplyr` functions that you will use to do the vast majority of data manipulations:

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

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

All verbs work similarly:

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

Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

6.6 `filter()` subsets data row-wise (observations).

You will want to isolate bits of your data; maybe you want to just look at a single country or a few years. R calls this subsetting.

`filter()` is a function in `dplyr` that takes logical expressions and returns the rows for which all are `TRUE`.

Visually, we are doing this (thanks RStudio for your cheatsheet):

Remember your logical expressions from this morning? We'll use `<` and `==` here.

```
filter(gapminder, lifeExp < 29)
```

You can say this out loud: "Filter the gapminder data for life expectancy less than 29". Notice that when we do this, all the columns are returned, but just the rows that have the life expectancy less than 29. We've subsetted by row.

Let's try another: "Filter the gapminder data for the country Mexico".

```
filter(gapminder, country == "Mexico")
```

How about if we want two country names? We can't use the `==` operator here, because it can only operate on one thing at a time. We will use the `%in%` operator:

```
filter(gapminder, country %in% c("Mexico", "Peru"))
```

How about if we want Mexico in 2002? You can pass filter different criteria:

```
filter(gapminder, country == "Mexico", year == 2002)
```

6.7 Your turn

What is the mean life expectancy of Sweden? Hint: do this in 2 steps by assigning a variable and then using the `mean()` function.

Then, sync to Github.com (pull, stage, commit, push).

6.7.1 Answer

```
x <- filter(gapminder, country == "Sweden")
mean(x$lifeExp)
```

6.8 Meet the new pipe %>% operator

Before we go any further, we should exploit the new pipe operator that `dplyr` imports from the `magrittr` package by Stefan Bache. **This is going to change your data analytical life**. You no longer need to enact multi-operation commands by nesting them inside each other. And we won't need to make temporary variables like we did in the Sweden example above. This new syntax leads to code that is much easier to write and to read: it actually tells the story of your analysis.

Here's what it looks like: `%>%`. The RStudio keyboard shortcut: Ctrl + Shift + M (Windows), Cmd + Shift + M (Mac).

Let's demo then I'll explain:

```
gapminder %>% head()
```

This is equivalent to `head(gapminder)`. This pipe operator takes the thing on the left-hand-side and **pipes** it into the function call on the right-hand-side – literally, drops it in as the first argument.

Never fear, you can still specify other arguments to this function! To see the first 3 rows of Gapminder, we could say `head(gapminder, 3)` or this:

```
gapminder %>% head(3)
```

I've advised you to think "gets" whenever you see the assignment operator, <-. Similary, you should think "and then" whenever you see the pipe operator, %>%.

You are probably not impressed yet, but the magic will soon happen.

Fun break: check out this gif about %>% from Twitter.

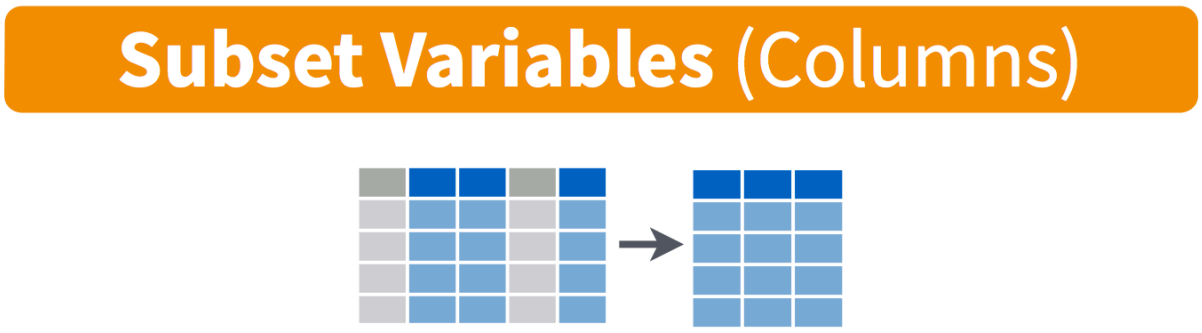


Figure 6.4:

6.9 `select()` subsets data column-wise (variables)

Back to `dplyr` ...

Use `select()` to subset the data on variables or columns.

Visually, we are doing this (thanks RStudio for your cheatsheet):

Here's a conventional call. Again, see that we can select multiple columns just with a comma, after we specify the data frame (gapminder).

```
select(gapminder, year, lifeExp)
```

But using what we just learned, with a pipe, we can do this:

```
gapminder %>% select(year, lifeExp)
```

Let's write it again but using multiple lines so it's nicer to read. And let's add a second pipe operator to pipe through `head`:

```
gapminder %>%
  select(year, lifeExp) %>%
  head(4)
```

Think: "Take `gapminder`, then select the variables year and lifeExp, then show the first 4 rows."

Being able to read a story out of code like this is really game-changing.

6.9.1 Revel in the convenience

Let's take the gapminder data and filter for the country Cambodia, and select 4 of the columns: country, year, pop, gdpPercap.

```
gapminder %>%
  filter(country == "Cambodia") %>%
  select(country, year, pop, gdpPercap)
```

But entering each column by hand can be tedious, especially since there are fewer columns we *don't* want. So instead, we can do:

```
gapminder %>%
  filter(country == "Cambodia") %>%
  select(-continent, -lifeExp) # you can use - to deselect columns
```

Figure 6.5:

6.10 mutate() adds new variables

Alright, let's keep going.

Let's say we needed to add an index column so we know which order these data came in. Let's not make a new variable, let's add a column to our gapminder data frame. How do we do that? With the `mutate()` function.

Visually, we are doing this (thanks RStudio for your cheatsheet):

We will name our new column index. We will name the new column 'index'; and we assign it with a single `=`. Notice that we can use the `nrow` function *within* our mutate call:

```
gapminder %>%
  mutate(index = 1:nrow(gapminder))
```

OK, let's do another example. Imagine we wanted to recover each country's GDP. After all, the Gapminder data has a variable for population and GDP per capita.

```
gapminder %>%
  mutate(gdp = pop * gdpPercap)
```

6.10.1 Your turn

Find the maximum gdpPercap of Egypt and Vietnam Create a new column.

Then, sync to Github.com (pull, stage, commit, push).

6.10.1.1 Answer

```
gapminder %>%
  select(-continent, -lifeExp) %>% # not super necessary but to simplify
  filter(country == "Egypt") %>%
  mutate(gdp = pop * gdpPercap) %>%
  mutate(max_gdp = max(gdp))

## you can also create multiple variables within the same mutate(), and line them up so they are easier
gapminder %>%
  select(-continent, -lifeExp) %>% # not super necessary but to simplify
  filter(country == "Vietnam") %>%
```

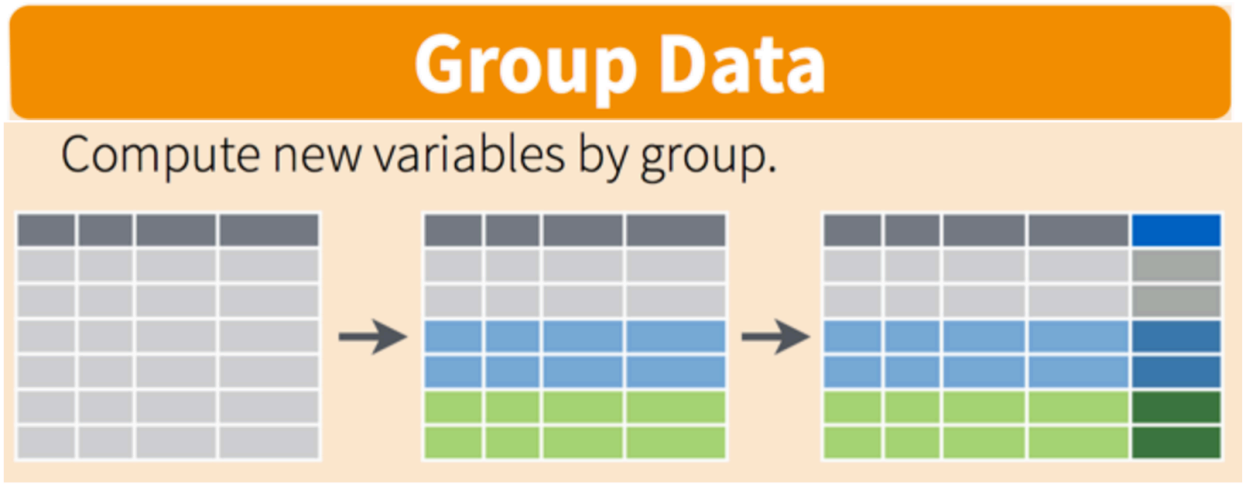


Figure 6.6:

```
  mutate(gdp      = pop * gdpPercap,
         max_gdp = max(gdp))
```

With the things we know so far, the answers you have are maybe a bit limiting. First, We had to act on Egypt and Vietnam separately, and repeat the same code. Copy-pasting like this is also super error prone.

And second, this `max_gdpPercap` column is pretty redundant, because it's a repeated value a ton of times. Sometimes this is exactly what you want! You are now set up nicely to maybe take a proportion of gdpPercap/max_gdpPercap for each year or something. But maybe you just wanted that max_gdpPercap for something else. Let's keep going…

6.11 `group_by()` operates on groups

Let's tackle that first issue first. So how do we less painfully calculate the max gdpPercap for all countries?

Visually, we are doing this (thanks RStudio for your cheatsheet):

```
gapminder %>%
  group_by(country) %>%
  mutate(gdp      = pop * gdpPercap,
         max_gdp = max(gdp)) %>%
  ungroup() # if you use group_by, also use ungroup() to save heartache later
```

So instead of filtering for a specific country, we've grouped by country, and then done the same operations. It's hard to see; let's look at a bunch at the tail:

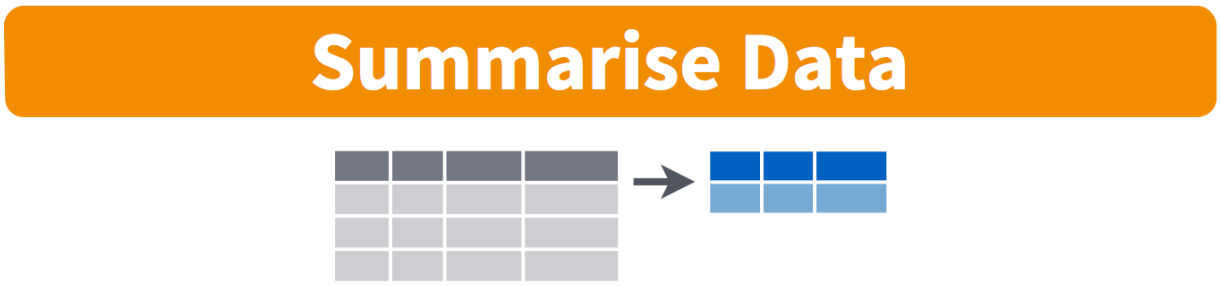
```
gapminder %>%
  group_by(country) %>%
  mutate(gdp      = pop * gdpPercap,
         max_gdp = max(gdp)) %>%
  ungroup() %>%
  tail(30)
```

OK, this is great. But what if this what we needed, a max_gdp value for each country. We don't need that kind of repeated value for each of the max_gdp values. Here's the next function:

6.11.1 summarize() with group_by()

We want to operate on a group, but actually collapse or distill the output from that group. The `summarize()` function will do that for us.

Visually, we are doing this (thanks RStudio for your cheatsheet):



Here we go:

```
gapminder %>%
  group_by(country) %>%
  mutate(gdp = pop * gdpPercap) %>%
  summarize(max_gdp = max(gdp)) %>%
  ungroup()
```

How cool is that! `summarize()` will actually only keep the columns that are grouped_by or summarized. So if we wanted to keep other columns, we'd have to do it another way (we'll get into it tomorrow).

6.12 arrange() orders columns

This is ordered alphabetically, which is cool. But let's say we wanted to order it in ascending order for `max_gdp`. The dplyr function is `arrange()`.

```
gapminder %>%
  group_by(country) %>%
  mutate(gdp = pop * gdpPercap) %>%
  summarize(max_gdp = max(gdp)) %>%
  ungroup() %>%
  arrange(max_gdp)
```

6.12.1 Your turn

1. arrange your data frame in descending order (opposite of what we've done). Expect that this is possible: `?arrange`
2. save your data frame as a variable
3. find the maximum life expectancy for countries in Asia. What is the earliest year you encounter? The latest? Hint: you can use or `base::max` and `dplyr::arrange()`…
4. Knit your RMarkdown file, and sync it to GitHub (pull, stage, commit, push)

6.12.1.1 Answer (no peeking!)

```
gapminder %>%
  filter(continent == 'Asia') %>%
  group_by(country) %>%
  filter(lifeExp == max(lifeExp)) %>%
  arrange(year)
```

6.13 All together now

We have done a pretty incredible amount of work in a few lines. Our whole analysis is this. Imagine the possibilities from here. It's very readable: you see the data as the first thing, it's not nested. Then, you can read the verbs. This is the whole thing, with explicit package calls from `readr::` and `dplyr::`:

```
## gapminder-wrangle.R
## J. Lowndes lowndes@nceas.ucsb.edu


## load libraries
library(tidyverse) ## install.packages('tidyverse')

## read in data
gapminder <- readr::read_csv('https://raw.githubusercontent.com/jules32/2017-11-30-MBARI/gh-pages/data/g

## summarize
max_gdp <- gapminder %>%
  dplyr::select(-continent, -lifeExp) %>% # or select(country, year, pop, gdpPercap)
  dplyr::group_by(country) %>%
  dplyr::mutate(gdp = pop * gdpPercap) %>%
  dplyr::summarize(max_gdp = max(gdp)) %>%
  dplyr::ungroup()
```

I actually am borrowing this "All together now" from Tony Fischetti's blog post How dplyr replaced my most common R idioms). With that as inspiration, this is how what we have just done would look like in Base R.

6.13.1 Compare to base R

Let's compare with some base R code to accomplish the same things. Base R requires subsetting with the `[rows, columns]` notation. This notation is something you'll see a lot in base R. the brackets `[ ]` allow you to extract parts of an object. Within the brackets, the comma separates rows from columns.

If we don't write anything after the comma, that means "all columns". And if we don't write anything before the comma, that means "all rows".

Also, the `$` operator is how you access specific columns of your dataframe. You can also add new columns like we do with `mex$gdp`.

Here we will just calculate the max for one country, Mexico. Tomorrow we will learn how to do it for all the countries, like we did with `dplyr::group_by()`.

```
## gapminder-wrangle.R --- baseR
## J. Lowndes lowndes@nceas.ucsb.edu
```

```
## Note the stringAsFactors = FALSE variable to avoid factors!
gapminder <- read.csv('https://raw.githubusercontent.com/jules32/2017-11-30-MBARI/gh-pages/data/gapmind

## subsetting columns. Compare to `dplyr::select()`
x1  <- gapminder[ , c('country', 'year', 'pop', 'gdpPercap') ]

## subsetting rows. Compare to `dplyr::filter()`
mex <- x1[x1$country == "Mexico", ]

## adding new columns. Compare to `dplyr::mutate()`.
mex$gdp <- mex$pop * mex$gdpPercap
mex$max_gdp <- max(mex$gdp)
```

Note too that the chain operator `%>%` that we used with the `tidyverse` lets us get away from the temporary variable x1.

6.13.2 Your Turn

Get your RMarkdown file cleaned up and sync it for the last time today!

6.13.2.1 Answers

...

6.14 Key Points

- Data manipulation functions in `dplyr` allow you to `filter()` by rows and `select()` by columns, create new columns with `mutate()`, and `group_by()` unique column values to apply `summarize()` for new columns that define aggregate values across groupings.
- The "then" operator `%>%` allows you to chain successive operations without needing to define intermediary variables for creating the most parsimonious, easily read analysis.

Chapter 7

Data Wrangling: `tidyr`

7.1 Overview

Now you have some experience wrangling and working with tidy data. But we all know that not all data that you have are tidy. So how do we make data more tidy? With `tidyr`.

Objectives

- learn `tidyr` with the `gapminder` package
- other wrangling: joins, binding
- practice the RStudio-GitHub workflow
- your turn: use the data wrangling cheat sheet to explore window functions

Resources

These materials borrow heavily from:

- R for Data Science: Relational Data
- R for Data Science: Tidy Data

7.2 `tidyr` basics

Often, data must be reshaped for it to become tidy data. What does that mean? There are four main verbs we'll use, which are essentially pairs of opposites:

- turn columns into rows (`gather()`),
- turn rows into columns (`spread()`),
- turn a character column into multiple columns (`separate()`),
- turn multiple character columns into a single column (`unite()`)

You use `spread()` and `gather()` to transform or reshape data between *wide* to *long* formats. *long* format is the tidy data we are after, where:

- each column is a variable
- each row is an observation

In the *long* format, you usually have 1 column for the observed variable and the other columns are ID variables.

For the *wide* format each row is often a site/subject/patient and you have multiple observation variables containing the same type of data. These can be either repeated observations over time, or observation of multiple variables (or a mix of both). Data input may be simpler or some other applications may prefer the 'wide' format. However, many of `R`'s functions have been designed assuming you have 'long' format data.

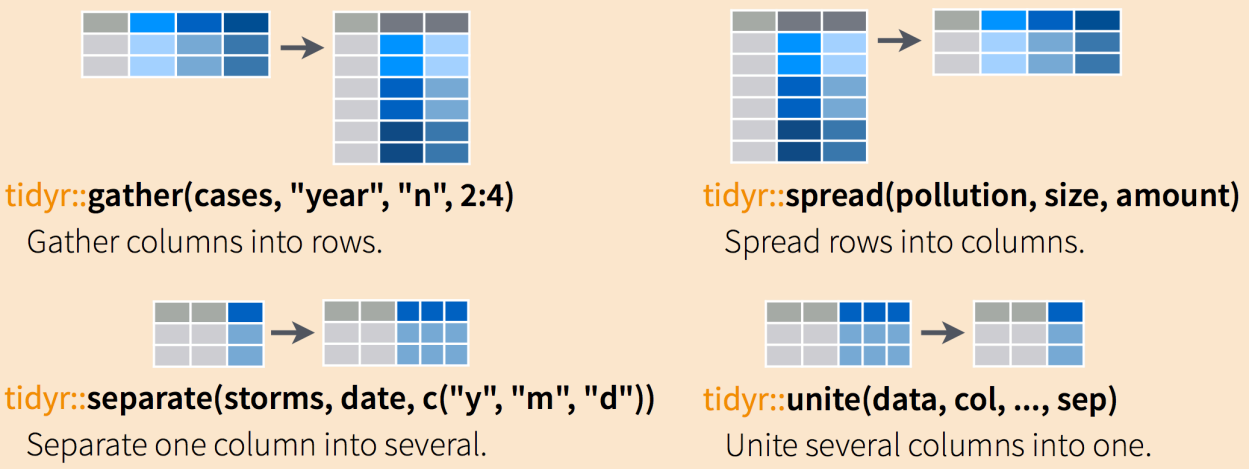


Figure 7.1:

These data formats mainly affect readability. For humans, the wide format is often more intuitive since we can often see more of the data on the screen due to it's shape. However, the long format is more machine readable and is closer to the formatting of databases. The ID variables in our dataframes are similar to the fields in a database and observed variables are like the database values.

> Question: Is gapminder a purely long, purely wide, or some intermediate format?

Sometimes, as with the gapminder dataset, we have multiple types of observed data. It is somewhere in between the purely 'long' and 'wide' data formats:

- 3 "ID variables" (`continent`, `country`, `year`)
- 3 "Observation variables" (`pop`,`lifeExp`,`gdpPercap`).

It's pretty common to have data in this intermediate format in most cases despite not having ALL observations in 1 column, since all 3 observation variables have different units. But we can play with switching it to long format and wide to show what that means (i.e. long would be 4 ID variables and 1 observation variable).

Note: Generally, mathematical operations are better in long format, although some plotting functions actually work better with wide format.

7.2.1 Setup

We'll work today in RMarkdown. You can either continue from the same RMarkdown as yesterday, or begin a new one.

Here's what to do:

1. Clear your workspace (Session > Restart R)
2. New File > R Markdown..., save as something other than `gapminder-wrangle.Rmd` and delete irrelevant info, or just continue using `gapminder-wrangle.Rmd`

I'm going to write this in my R Markdown file:

```
Data wrangling with `tidyr`, which is part of the tidyverse. We are going to tidy some data!
```

continent	country	gdpPercap_1952	gdpPercap_1957	gdpPercap_1962	gdpPercap_1967	gdpPercap_1972	gdpPercap_1977
Africa	Algeria	2449.008185	3013.976023	2550.81688	3246.991771	4182.663766	4910.416756
Africa	Angola	3520.610273	3827.940465	4269.276742	5522.776375	5473.288005	3008.647355
Africa	Benin	1062.7522	959.6010805	949.4990641	1035.831411	1085.796879	1029.161251
Africa	Botswana	851.2411407	918.2325349	983.6539764	1214.709294	2263.611114	3214.857818
Africa	Burkina Faso	543.2552413	617.1834648	722.5120206	794.8265597	854.7359763	743.3870368
Africa	Burundi	339.2964587	379.5646281	355.2032273	412.9775136	464.0995039	556.1032651
Africa	Cameroon	1172.667655	1313.048099	1399.607441	1508.453148	1684.146528	1783.432873
Africa	Central African Republic	1071.310713	1190.844328	1193.068753	1136.056615	1070.013275	1109.374338
Africa	Chad	1178.665927	1308.495577	1389.817618	1196.810565	1104.103987	1133.98495
Africa	Comoros	1102.990936	1211.148548	1406.648278	1876.029643	1937.577675	1172.603047
Africa	Congo Dem. Rep.	780.5423257	905.8602303	896.3146335	861.5932424	904.8960685	795.757282
Africa	Congo Rep.	2125.621418	2315.056572	2464.783157	2677.939642	3213.152683	3259.178978
Africa	Cote d'Ivoire	1388.594732	1500.895925	1728.869428	2052.050473	2378.201111	2517.736547

Figure 7.2:

7.2.2 load `tidyverse` (which has `tidyr` inside)

First load `tidyr` in an R chunk. You already have installed the tidyverse, so you should be able to just load it like this (using the comment so you can run `install.packages("tidyverse")` easily if need be):

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

7.3 Explore gapminder data — wide format.

Yesterday we started off with the gapminder data in a format that was already tidy. But what if it weren't? Let's look at a different version of those data.

The data are on GitHub. Navigate there by going to:

github.com > ohi-science > data-science-training > data > gapminder_wide.csv

or by copy-pasting this in the browser: `https://github.com/OHI-Science/data-science-training/blob/master/data/ga`

Have a look at the data. You can see there are a lot more columns than the version we looked at before. This format is pretty common, because it can be a lot more intuitive to *enter* data in this way.

But we want it to be in a tidy way so that we can work with it more easily. So here we go.

7.4 `gather()` data from wide to long format

Read in the data from GitHub. Remember, you need to click on the 'Raw' button first so you can read it directly. Let's also read in the gapminder data from yesterday so that we can use it to compare later on.

```
## wide format
gap_wide <- readr::read_csv('https://raw.githubusercontent.com/OHI-Science/data-science-training/master/
```

```
## yesterday's format (intermediate)
gapminder <- readr::read_csv('https://raw.githubusercontent.com/OHI-Science/data-science-training/maste
```

Let's have a look:

```
head(gap_wide)
str(gap_wide)
```

While wide format is nice for data entry, it's not nice for calculations. Some of the columns are a mix of variable (e.g. "gdpPercap") and data ("1952"). What if you were asked for the mean population after 1990 in Algeria? Possible, but ugly. But we know it doesn't need to be so ugly. Let's tidy it back to the format we've been using.

> Question: let's talk this through together. If we're trying to turn the `gap_wide` format into `gapminder` format, what structure does it have that we like? And that we want to change?

- We like the continent and country columns. We won't want to change those.
- For long format, we'd want just 1 column identifying the variable name (`tidyr` calls this a **'key'**), and 1 column for the data (`tidyr` calls this the '**value'**).
- For intermediate format, we'd want 3 columns for `gdpPercap`, `lifeExp`, and `pop`.
- We would like year as a separate column.

Let's get it to long format. We'll have to do this in 2 steps. The first step is to take all of those column names (e.g. `lifeExp_1970`) and make them a variable in a new column, and transfer the values into another column. Let's learn by doing:

Let's have a look at `gather()`'s help:

```
?gather
```

> Question: What is our **key-value pair**?

We need to name two new variables in the key-value pair, one for the key, one for the value. It can be hard to wrap your mind around this, so let's give it a try. Let's name them `obstype_year` and `obs_value`.

Here's the start of what we'll do:

```
gap_long <- gap_wide %>%
  gather(key   = obstype_year,
         value = obs_values)
```

We got a warning message. This means that `gather()` worked, but maybe not how we wanted it to to.

Although we were already planning to inspect our work, let's definitely do it now:

```
str(gap_long)
head(gap_long)
tail(gap_long)
```

So we have successfully reshaped our dataframe, but really not how we wanted. Very important to check, and listen to that warning message–dropping attributes seems very suspicious.

What went wrong? Notice that it didn't know that we wanted to keep `continent` and `country` untouched; we need to give it more information about which columns we want reshaped. We can do this in several ways.

A good way: identify the columns by name. Listing them out by explicit name can be a good approach if there are a few. But there's a lot here: over 30. But I'm not going to list them out here, and way too much potential for error if you tried gdpPercap_1952, gdpPercap_1957, gdpPercap_1962... But we could use some of `dplyr`'s awesome helper functions — because we expect that there is a better way to do this!

```
gap_long <- gap_wide %>%
  gather(key   = obstype_year,
         value = obs_values,
```

```
         dplyr::starts_with('pop'),
         dplyr::starts_with('lifeExp'),
         dplyr::starts_with('gdpPercap'))

str(gap_long)
head(gap_long)
tail(gap_long)
```

Success! And there is another way that is nice to use if your columns don't follow such a structured pattern: you can exclude the columns you *don't* want.

```
gap_long <- gap_wide %>%
  gather(key   = obstype_year,
         value = obs_values,
         -continent, -country)

str(gap_long)
head(gap_long)
tail(gap_long)
```

To recap:

Inside `gather()` we first name the new column for the new ID variable (`obstype_year`), the name for the new amalgamated observation variable (`obs_value`), then the names of the old observation variable. We could have typed out all the observation variables, but as in the `select()` function (see `dplyr` lesson), we can use the `starts_with()` argument to select all variables that starts with the desired character string. Gather also allows the alternative syntax of using the `-` symbol to identify which variables are not to be gathered (i.e. ID variables).

OK, but we're not done yet. `obstype_year` actually contains two pieces of information, the observation type (`pop`,`lifeExp`, or `gdpPercap`) and the `year`. We can use the `separate()` function to split the character strings into multiple variables.

`?separate` –> the main arguments are `separate(data, col, into, sep ...)`. So we need to specify which column we want separated, name the new columns that we want to create, and specify what we want it to separate by. Since the `obstype_year` variable has observation types and years separated by a `_`, we'll use that.

```
gap_long <- gap_wide %>%
  gather(key   = obstype_year,
         value = obs_values,
         -continent, -country) %>%
  separate(obstype_year,
           into = c('obs_type','year'),
           sep="_")
```

No warning messages...still we inspect:

```
str(gap_long)
head(gap_long)
tail(gap_long)
```

Excellent. This is long format: every row is a unique observation. Yay!

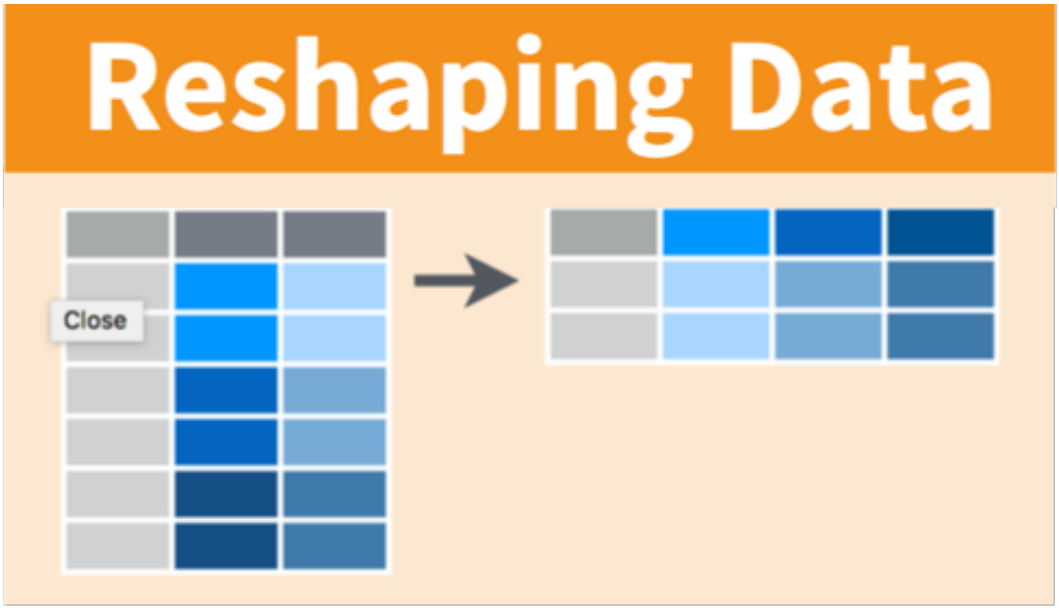


Figure 7.3:

7.4.1 Your turn

1. Using `gap_long`, calculate the mean life expectancy, population, and gdpPercap for each continent. **Hint:** use the `dplyr::group_by()` and `dplyr::summarize()` functions
2. What other helper functions can you use with `dplyr::select()`? Would any be useful in our example above? Why or why not?
3. Knit the R Markdown file and sync to Github (pull, stage, commit, push)

```
# solution (no peeking!)
gap_long %>%
  group_by(continent, obs_type) %>%
    summarize(means = mean(obs_values))
```

7.5 `spread()` data from long to intermediate format

Alright! Now just to double-check our work, let's use the opposite of `gather()` to spread our observation variables back to the original format with the aptly named `spread()`. You pass `spread()` the key and value pair, which is now `obs_type` and `obs_values`.

```
gap_normal <- gap_long %>%
  spread(obs_type, obs_values)
```

No warning messages is good...but still let's check:

```
dim(gap_normal)
dim(gapminder)
names(gap_normal)
names(gapminder)
```

Now we've got an intermediate dataframe `gap_normal` with the same dimensions as the original `gapminder`, but the order of the variables is different. Let's fix that before checking if they are `all.equal()`.

7.5.1 Your turn

Reorder the columns in "gap_normal" to match "gapminder".

7.5.1.1 Answer (no peeking!)

```
# one way with dplyr and %>%
gap_normal <- gap_normal %>%
  select(country, continent, year, lifeExp, pop, gdpPercap)

# another way with base R
gap_normal <- gap_normal[,names(gapminder)]
```

Now let's check if they are all.equal (`?all.equal`) is a handy test

```
all.equal(gap_normal, gapminder)
```

Hmm. Our `all.equal()` test didn't pass. Let's try to figure out why:

```
head(gap_normal)
head(gapminder)
```

Ah, they are ordered differently. We're almost there, the original was ordered by `country`, `continent`, then `year`.

```
gap_normal <- gap_normal %>%
  arrange(country, continent, year)

all.equal(gap_normal, gapminder)
```

Better…

```
str(gap_normal)
str(gapminder)
```

Mine currently shows that the in gapminder, "year" is an integer (int), but in gap_normal, "year" is a character. So let's change that and see if that helps:

```
gap_normal <- gap_normal %>%
  mutate(year = as.integer(year))

all.equal(gap_normal, gapminder)
```

Hooray!

```
str(gap_normal)
str(gapminder)
```

(In the past, mine has shown a slight difference because one is a data.frame and one is a tbl_df, which is similar to a data.frame. We won't get into this difference now, I'm feeling good about these data sets! We've gone from the longest format back to the intermediate and we didn't introduce any errors in our code.)

7.6 Your turn

1. Convert "gap_long" all the way back to gap_wide. Hint: you'll need to create appropriate labels for all our new variables (time*metric combinations) with the opposite of separate: `tidyr::unite()`.
2. Knit the R Markdown file and sync to Github (pull, stage, commit, push)

7.6.1 Answer (no peeking)

```
head(gap_long) # remember the columns

gap_wide_new <- gap_long %>%
  # first unite obs_type and year into a new column called var_names. Separate by _
  unite(col = var_names, obs_type, year, sep = "_") %>%
  # then spread var_names out by key-value pair.
  spread(key = var_names, value = obs_values)
str(gap_wide_new)
```

7.6.2 clean up and save your .Rmd

Spend some time cleaning up and saving `gapminder-wrangle.Rmd` Restart R. In RStudio, use *Session > Restart R*. Otherwise, quit R with `q()` and re-launch it.

This morning's .Rmd could look something like this:

```
## load tidyr (in tidyverse)
library(tidyverse) # install.packages("tidyverse")

## load wide data
gap_wide <- read.csv('https://raw.githubusercontent.com/OHI-Science/data-science-training/master/data/g

head(gap_wide)
str(gap_wide)

## practice tidyr::gather() wide to long
gap_long <- gap_wide %>%
  gather(key   = obstype_year,
         value = obs_values,
         -continent, -country)
# or
gap_long <- gap_wide %>%
  gather(key   = obstype_year,
         value = obs_values,
         dplyr::starts_with('pop'),
         dplyr::starts_with('lifeExp'),
         dplyr::starts_with('gdpPercap'))

## gather() and separate() to create our original gapminder
gap_long <- gap_wide %>%
  gather(key   = obstype_year,
         value = obs_values,
         -continent, -country) %>%
  separate(obstype_year,
```

```
            into = c('obs_type','year'),
            sep="_")

## practice: can still do calculations in long format
gap_long %>%
  group_by(continent, obs_type) %>%
  summarize(means = mean(obs_values))

## spread() from normal to wide
gap_normal <- gap_long %>%
  spread(obs_type, obs_values) %>%
  select(country, continent, year, lifeExp, pop, gdpPercap)

## check that all.equal()
all.equal(gap_normal,gapminder)

## unite() and spread(): convert gap_long to gap_wide
head(gap_long) # remember the columns

gap_wide_new <- gap_long %>%
  # first unite obs_type and year into a new column called var_names. Separate by _
  unite(col = var_names, obs_type, year, sep = "_") %>%
  # then spread var_names out by key-value pair.
  spread(key = var_names, value = obs_values)
str(gap_wide_new)
```

7.6.3 `complete`: other `tidyr` awesomeness

For this, let's look at Jarrett Byrnes' blog on the topic:

http://www.imachordata.com/you-complete-me/

7.7 Other links

- Tidying up Data - Env Info - Rmd
- Data wrangling with dplyr and tidyr - Tyler Clavelle & Dan Ovando - Rmd

Chapter 8

Programming

8.1 Objectives and Resources

Now we are going to build a little analysis. We will learn to automate our analyses with a for loop. We will make figs, save them each with automated labeling. Then, we will join data from different files and conditionally label them with if/else statements.

Ultimately, with our analysis, we want to plot…

Objectives

- discuss good file naming practices
- create an R script
- for loops
- joining data
- if statements
- make sure your loop worked like you wanted
- if statements (conditionals)
  - write message() to yourself
  - list.files()
- importing and writing data
  - write a local copy of gapminder data to data/ folder
  - installing packages from github

Resources

8.2 Naming files

Now is a good interlude to talk about naming things.

We are going to take five minutes to talk through Jenny Bryan's three principles for naming files:

1. machine readable
2. human readable
3. play well with default ordering

8.3 Analysis plan

OK, here is the plan for our analysis. We want to plot the gdpPercap for each country in the gapminder data frame. We will label each one and save it in a folder called figures. We will learn a bunch of things as we go.

8.4 Create an R script

OK, now, we are going to create an R script. What is an R script? It's a text file with a .R extension. We've been writing R code in R Markdown files so far; R scripts are just R code without the Markdown along with it.

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

Let's start off with a few comments so that we know what it is for, and save it:

```
## gapminder-analysis.R
## analysis with gapminder data
## J Lowndes lowndes@nceas.uscb.edu
```

We'll be working with the gapminder data again so let's read it in here:

```
## load libraries
library(tidyverse)

## read in gapminder data
gapminder <- readr::read_csv('https://raw.githubusercontent.com/OHI-Science/data-science-training/maste
```

Remember, like in R Markdown, hitting return does not execute this command. To execute it, we need to get what we typed in the script down into the console. Here is how we can do that:

1. copy-paste this line into the console.
2. select the line (or simply put the cursor there), and click 'Run'. This is available from
    a. the bar above the script (green arrow)
    b. the menu bar: Code > Run Selected Line(s)
    c. keyboard shortcut: command-return
3. source the script, which means running the whole thing. This is also great for to see if there are any typos in your code that you've missed. You can do this by:
    a. clicking Source (blue arrow in the bar above the script).
    b. typing `source('gapminder-analysis.R')` in the console (or from another R file!!!).

8.5 Automation with for loops

Our plan is to plot gdpPercap for each country. This means that we want to do the same operation (plotting gdpPercap) on a bunch of different things (countries). Yesterday we learned the dplyr's `group_by()` function, and this is super powerful to automate through groups. But there are things that `group_by()` can't do, like plotting. So we will use a for loop.

Let's start off with what this would look like for just one country. I'm going to demonstrate with Afghanistan:

```
## filter the country to plot
gap_to_plot <- gapminder %>%
  filter(country == "Afghanistan")

## plot
my_plot <- ggplot(data = gap_to_plot, aes(x = year, y = gdpPercap)) +
```

```
  geom_point() +
  labs(title = "Afghanistan")
```

Let's actually give this a better title than just the country name. Let's use the `base::paste()` function from to paste two strings together so that the title is more descriptive. Use `?paste` to see what the "sep" variable does.

```
## filter the country to plot
gap_to_plot <- gapminder %>%
  filter(country == "Afghanistan")

## plot
my_plot <- ggplot(data = gap_to_plot, aes(x = year, y = gdpPercap)) +
  geom_point() +
  ## add title and save
  labs(title = paste("Afghanistan", "GDP per capita", sep = " "))
```

And as a last step, let's save this figure using `base::file.path()` (which works like paste() would if sep = "/").

```
## filter the country to plot
gap_to_plot <- gapminder %>%
  filter(country == "Afghanistan")

## plot
my_plot <- ggplot(data = gap_to_plot, aes(x = year, y = gdpPercap)) +
  geom_point() +
## add title and save
  labs(title = paste("Afghanistan", "GDP per capita", sep = " "))

ggsave(filename = "Afghanistan_gdpPercap.png", plot = my_plot,
         width = 15, height = 10)
```

OK. So we can check in our figures/folder and see the generated figure:

And there wasn't that much code needed to get us here, but we definitely do not want to copy this for every country. Even if we copy-pasted and switched out the names, it would be very typo-prone. Plus, what if you wanted to instead plot lifeExp? You'd have to remember to change it each time...it gets messy quick.

Better with a for loop. This will let us cycle through and do what we want to each thing in turn. If you want to iterate over a set of values, and perform the same operation on each, a `for` loop will do the job.

8.5.1 For loop basic structure

The basic structure of a `for` loop is:

```
for( each item in set of items ){
  do a thing
}
```

Note the `( )` and the `{ }`. We talk about iterating through each item in the for loop, which makes each item an iterator.

So looking back at our Afghanistan code: all of this is pretty much the "do a thing" part. And we can see that there are only a few places that are specific to Afghanistan. If we could make those places not specific to Afghanistan, we would be set.

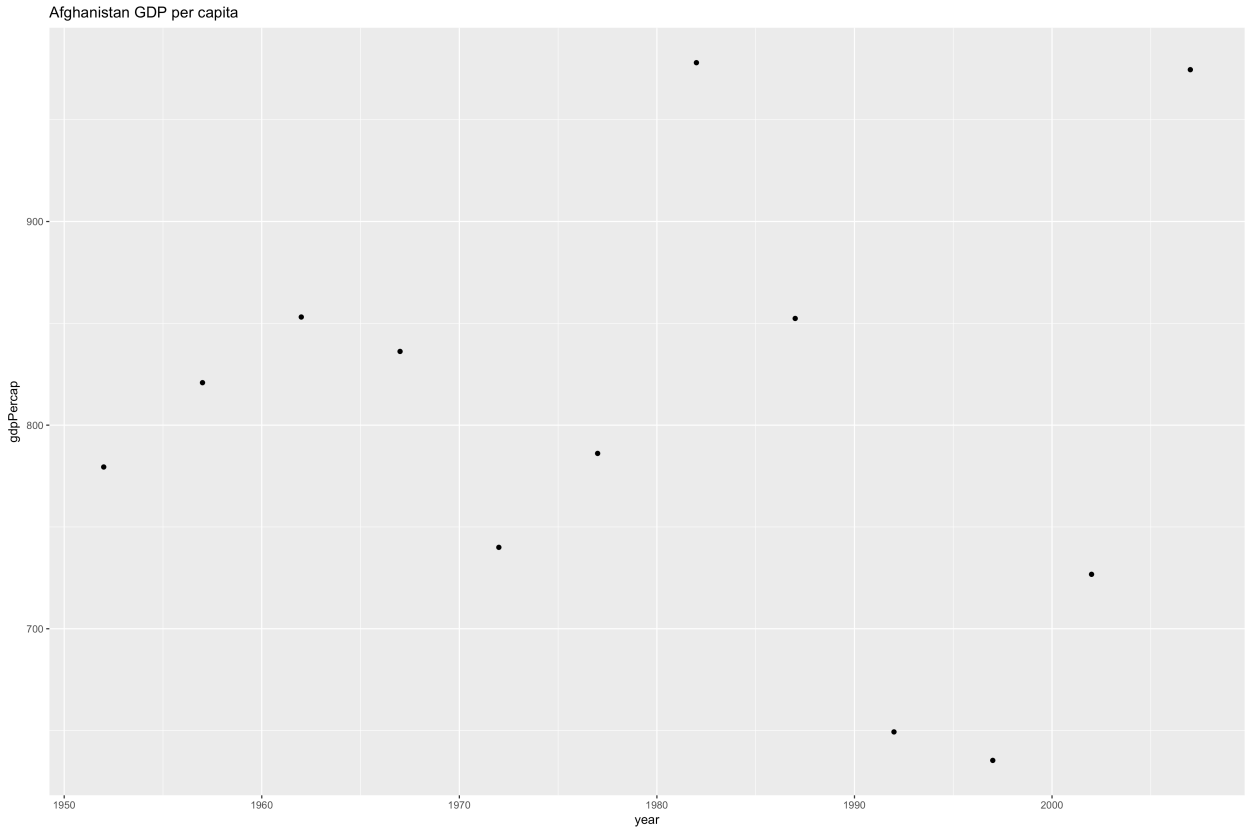


Figure 8.1:

```
for (each country in a set of countries) {
## filter the country to plot
gap_to_plot <- gapminder %>%
  filter(country == "Afghanistan")
a country

## plot
ggplot(data = gap_to_plot, aes(x = year, y = gdpPercap)) +
  geom_point() +
## add title and save
a country
  labs(title = paste("Afghanistan", "GDP per capita", sep = " ")) +
  ggsave(filename = file.path("figures", "Afghanistan_gdpPercap.png"),
         width = 15, height = 10)
a country
}
```

Figure 8.2:

Let's paste from what we had before, and modify it. I'm also going to use RStudio's indentation help to indent the lines within the for loop by highlighting the code in this chunk and going to Code > Reindent Lines (shortcut: command I)

```
for( each item in set of items ){

  ## filter the country to plot
  gap_to_plot <- gapminder %>%
    filter(country == "Afghanistan")

  ## plot
  my_plot <- ggplot(data = gap_to_plot, aes(x = year, y = gdpPercap)) +
    geom_point() +
    ## add title and save
    labs(title = paste("Afghanistan", "GDP per capita", sep = " "))

  ggsave(filename = "Afghanistan_gdpPercap.png", plot = my_plot
           width = 15, height = 10)
}
```

OK. So let's start with the beginning of the for loop. We want a list of countries that we will iterate through. We can do that by adding this code before the for loop. And we will need to name the iterator something, so let's call it cntry so that it has a distinct name.

We can also add a print statement so that we can watch it iterate:

```
## create a list of countries
country_list <- c("Albania", "Fiji", "Spain")

for( cntry in country_list ){

  ## filter the country to plot
  gap_to_plot <- gapminder %>%
    filter(country == "Afghanistan")

  ## add a print message
  print(paste("Plotting", cntry))

  ## plot
  my_plot <- ggplot(data = gap_to_plot, aes(x = year, y = gdpPercap)) +
    geom_point() +
    ## add title and save
    labs(title = paste("Afghanistan", "GDP per capita", sep = " "))

ggsave(filename = "Afghanistan_gdpPercap.png", plot = my_plot,
           width = 15, height = 10)
}
```

At this point, we do have a functioning for loop. For each item in the `country_list$country`, the for loop will iterate over the code within the `{ }`, changing `cntry` each time as it goes through the list. And we can see it works because our print statement displays each country.

But our code doesn't work the way we expected. Why? Well, is looping through the 3 countries in our country_list, but it is creating plots for Afghanistan each time. We can see that by looking in the git tab: only that one Afghanistan figure. It's because we haven't brought the `cntry` variable into the for loop. Let's do that now.

8.5.2 Executable for loop!

```
## create a list of countries
country_list <- c("Albania", "Fiji", "Spain")

for( cntry in country_list ){

  ## filter the country to plot
  gap_to_plot <- gapminder %>%
    filter(country == cntry)

  ## add a print message
  print(paste("Plotting", cntry))

  ## plot
  my_plot <- ggplot(data = gap_to_plot, aes(x = year, y = gdpPercap)) +
    geom_point() +
    ## add title and save
    labs(title = paste(cntry, "GDP per capita", sep = " "))

ggsave(filename = paste(cntry, "_gdpPercap.png", sep = ""), plot = my_plot,
           width = 15, height = 10)
}
```

Great! And it doesn't matter if we just use these three countries or all the countries–let's try it.

But first let's create a figure directory and make sure it saves there since it's going to get out of hand quickly. We could do this from the Finder/Windows Explorer, or from the "Files" pane in RStudio by clicking "New Folder" (green plus button). But we are going to do it in R. A folder is called a directory:

```
dir.create("figures")

## create a list of countries
country_list <- unique(gapminder$country) # ?unique() returns the unique values

for( cntry in country_list ){

  ## filter the country to plot
  gap_to_plot <- gapminder %>%
    filter(country == cntry)

  ## add a print message
  print(paste("Plotting", cntry))

  ## plot
  my_plot <- ggplot(data = gap_to_plot, aes(x = year, y = gdpPercap)) +
    geom_point() +
    ## add title and save
    labs(title = paste(cntry, "GDP per capita", sep = " "))

ggsave(filename = paste("figures/", cntry, "_gdpPercap.png", sep = ""), plot = my_plot,
           width = 15, height = 10)
}
```

So that took a little longer than just the 3, but still super fast. For loops are sometimes just the thing you

need to iterate over many things in your analyses.

8.5.3 Clean up our repo

OK we now have 142 figures that we just created. They exist locally on our computer, and we have the code to recreate them anytime. But, we don't really need to push them to GitHub. Let's delete the figures/ folder and see it disappear from the Git tab.

8.5.4 Your turn

1. Modify our for loop so that it:
   - loops through countries in Europe only
   - plots the cumulative mean gdpPercap (Hint: Use the Data Wrangling Cheatsheet!)
   - saves them to a new subfolder inside the (recreated) figures folder called "Europe".
2. Sync to GitHub

8.5.4.1 Answer

No peeking!

```
dir.create("figures")
dir.create("figures/Europe")

## create a list of countries. Calculations go here, not in the for loop
gap_europe <- gapminder %>%
  filter(continent == "Europe") %>%
  mutate(gdpPercap_cummean = dplyr::cummean(gdpPercap))

country_list <- unique(gap_europe$country) # ?unique() returns the unique values

for( cntry in country_list ){ # (cntry = country_list[1])

  ## filter the country to plot
  gap_to_plot <- gap_europe %>%
    filter(country == cntry)

  ## add a print message
  print(paste("Plotting", cntry))

  ## plot
  my_plot <- ggplot(data = gap_to_plot, aes(x = year, y = gdpPercap_cummean)) +
    geom_point() +
    ## add title and save
    labs(title = paste(cntry, "GDP per capita", sep = " "))

ggsave(filename = paste("figures/Europe", cntry, "_gdpPercap_cummean.png", sep = ""),
       plot = my_plot, width = 15, height = 10)
}
```

Notice how we put the calculation for `cummean()` outside the for loop. It could have gone inside, but it's an operation that could be done just one time before hand (outside the loop) rather than multiple times as you go (inside the for loop).

8.6 Conditional statements with `if` and `else`

Often when we're coding we want to control the flow of our actions. This can be done by setting actions to occur only if a condition or a set of conditions are met.

In R and other languages, these are called "if statements".

8.6.1 if statement basic structure

```
# if
if (condition is true) {
  do something
}

# if ... else
if (condition is true) {
  do something
} else {  # that is, if the condition is false,
  do something different
}
```

Let's bring this concept into our for loop for Europe that we've just done. What if we want to add the label "Estimated" to countries that were estimated? Here's what we'd do.

```
dir.create("figures")
dir.create("figures/Europe")

## create a list of countries
gap_europe <- gapminder_est %>% ## use instead of gapminder
  filter(continent == "Europe") %>%
  mutate(gdpPercap_cummean = dplyr::cummean(gdpPercap))

country_list <- unique(gap_europe$country)

for( cntry in country_list ){ # (cntry = country_list[1])

  ## filter the country to plot
  gap_to_plot <- gap_europe %>%
    filter(country == cntry)

  ## add a print message
  print(paste("Plotting", cntry))

  ## plot
  my_plot <- ggplot(data = gap_to_plot, aes(x = year, y = gdpPercap_cummean)) +
    geom_point() +
    ## add title and save
    labs(title = paste(cntry, "GDP per capita", sep = " "))

  ## if estimated, add that as a subtitle.
  if (gap_to_plot$estimated == "yes") {

    ## add a print statement just to check
    print(paste(cntry, "data are estimated"))
```

```
    my_plot <- my_plot +
      labs(sutbtitle("Estimated data"))
  }
#   Warning message:
# In if (gap_to_plot$estimated == "yes") { :
#   the condition has length > 1 and only the first element will be used

  ggsave(filename = paste("figures/Europe", cntry, "_gdpPercap_cummean.png", sep = ""),
       plot = my_plot, width = 15, height = 10)

}
```

This worked, but we got a warning message with the if statement. This is because if we look at `gap_to_plot$estimated`, it is many "yes"s or "no"s, and the if statement works just on the first one. We know that if any are yes, all are yes, but you can imagine that this could lead to problems down the line if you *didn't* know that. So let's be explicit: ### Executable if statement

```
dir.create("figures")
dir.create("figures/Europe")

## create a list of countries
gap_europe <- gapminder_est %>% ## use instead of gapminder
  filter(continent == "Europe") %>%
  mutate(gdpPercap_cummean = dplyr::cummean(gdpPercap))

country_list <- unique(gap_europe$country)

for( cntry in country_list ){ # (cntry = country_list[1])

  ## filter the country to plot
  gap_to_plot <- gap_europe %>%
    filter(country == cntry)

  ## add a print message
  print(paste("Plotting", cntry))

  ## plot
  my_plot <- ggplot(data = gap_to_plot, aes(x = year, y = gdpPercap_cummean)) +
    geom_point() +
    ## add title and save
    labs(title = paste(cntry, "GDP per capita", sep = " "))

  ## if estimated, add that as a subtitle.
  if (any(gap_to_plot$estimated == "yes")) { # any() will return a single TRUE or FALSE

    print(paste(cntry, "data are estimated"))

    my_plot <- my_plot +
      labs(subtitle = "Estimated data")
  }
  ggsave(filename = paste("figures/Europe", cntry, "_gdpPercap_cummean.png", sep = ""),
       plot = my_plot, width = 15, height = 10)

}
```

OK so this is working as we expect! But an if/else statement could make us extra sure that everything is working appropriately.

8.6.2 Executable if/else statement

```
dir.create("figures")
dir.create("figures/Europe")

## create a list of countries
gap_europe <- gapminder_est %>% ## use instead of gapminder
  filter(continent == "Europe") %>%
  mutate(gdpPercap_cummean = dplyr::cummean(gdpPercap))

country_list <- unique(gap_europe$country)

for( cntry in country_list ){ # (cntry = country_list[1])

  ## filter the country to plot
  gap_to_plot <- gap_europe %>%
    filter(country == cntry)

  ## add a print message
  print(paste("Plotting", cntry))

  ## plot
  my_plot <- ggplot(data = gap_to_plot, aes(x = year, y = gdpPercap_cummean)) +
    geom_point() +
    ## add title and save
    labs(title = paste(cntry, "GDP per capita", sep = " "))

  ## if estimated, add that as a subtitle.
  if (any(gap_to_plot$estimated == "yes")) { # any() will return a single TRUE or FALSE

    print(paste(cntry, "data are estimated"))

    my_plot <- my_plot +
      labs(subtitle = "Estimated data")
  } else {

    print(paste(cntry, "data are reported"))

  }
  ggsave(filename = paste("figures/Europe", cntry, "_gdpPercap_cummean.png", sep = ""),
       plot = my_plot, width = 15, height = 10)

}
```

So now we have a working for loop with conditional if/else statements that we could build from.

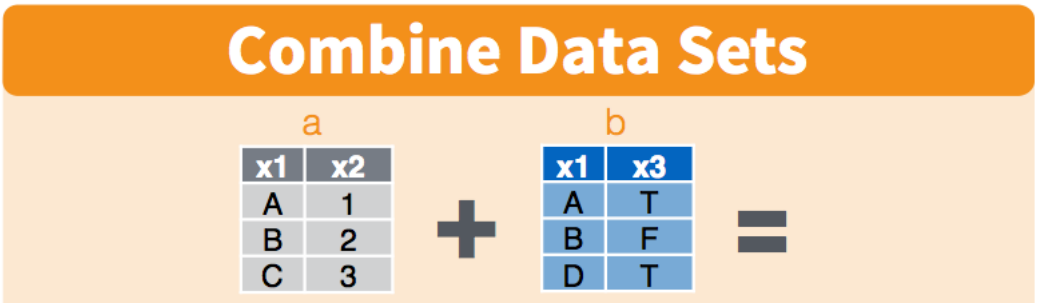


Figure 8.3:

8.7 Joining datasets

Let's say that our colleague just sent us another file that identifies when the data we're using was estimated versus measured (I'm making this up). Let's read in the file our colleague sent us:

```
## read in our colleague's data to join
countries_estimated <- gapminder <- readr::read_csv('https://raw.githubusercontent.com/OHI-Science/data-
```

```
## have a look
head(countries_estimated)
summary(countries_estimated)
str(countries_estimated)
```

(If you're interested, here is the code that I used to create this fake dataset:)

```
x <- countries_list %>%
  mutate(estimated = runif(length(country), 0, 1)) %>%
  mutate(estimated = round(estimated))

x$estimated[x$estimated == 0] = "no"
x$estimated[x$estimated == 1] = "yes"
readr::write_csv(x, 'data/countries_estimated.csv')
```

So, what can we see about these data. It looks like there are 142 countries represented, just like our gapminder data. There is only one entry (row) for each country. And we want to somehow join this data with the data we have from gapminder.

We want to do this:

8.7.1 Your turn

With a partner, look at RStudio's Data Wrangling Cheatsheet and explore the different ways that you could join data. Which way do you think would work here?

We are going to `left_join()` the gapminder data with the estimated data. Left joining the gapminder data will keep all the information in gapminder, but with additional information from the estimated data. So gapminder doesn't lose any rows.

Let's have a look:

```
## to check:
str(gapminder) # 1704 observations of 6 variables
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    142 obs. of  2 variables:
```

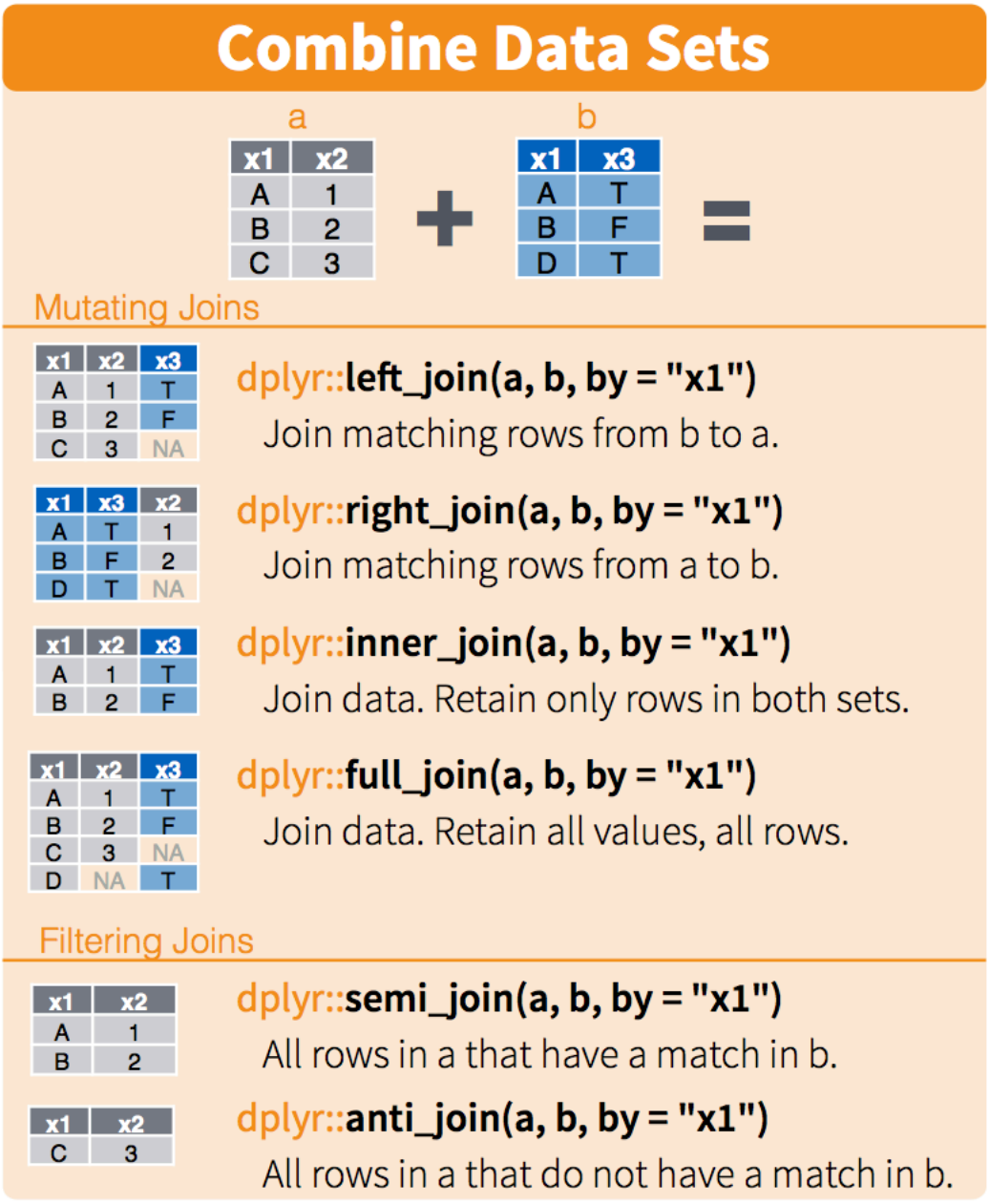

Combine Data Sets
a
x1 x2
A 1
B 2
C 3
b
x1 x3
A T
B F
D T
Mutating Joins
x1 x2 x3
A 1 T
B 2 F
C 3 NA
dplyr::left_join(a, b, by = "x1")
Join matching rows from b to a.
x1 x3 x2
A T 1
B F 2
D T NA
dplyr::right_join(a, b, by = "x1")
Join matching rows from a to b.
x1 x2 x3
A 1 T
B 2 F
dplyr::inner_join(a, b, by = "x1")
Join data. Retain only rows in both sets.
x1 x2 x3
A 1 T
B 2 F
C 3 NA
D NA T
dplyr::full_join(a, b, by = "x1")
Join data. Retain all values, all rows.
Filtering Joins
x1 x2
A 1
B 2
dplyr::semi_join(a, b, by = "x1")
All rows in a that have a match in b.
x1 x2
C 3
dplyr::anti_join(a, b, by = "x1")
All rows in a that do not have a match in b.

Figure 8.4:

```
##  $ country  : chr  "Afghanistan" "Albania" "Algeria" "Angola" ...
##  $ estimated: chr  "yes" "no" "yes" "no" ...
##  - attr(*, "spec")=List of 2
##   ..$ cols   :List of 2
##   .. ..$ country  : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ estimated: list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   ..$ default: list()
##   .. ..- attr(*, "class")= chr  "collector_guess" "collector"
##   ..- attr(*, "class")= chr "col_spec"
```

```
gapminder_est <- gapminder %>%
  left_join(countries_estimated,
            by = "country")

str(gapminder_est) #1704 obs. of  7 variables
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    142 obs. of  3 variables:
##  $ country    : chr  "Afghanistan" "Albania" "Algeria" "Angola" ...
##  $ estimated.x: chr  "yes" "no" "yes" "no" ...
##  $ estimated.y: chr  "yes" "no" "yes" "no" ...
```

```
head(gapminder_est, 20)
```

```
## # A tibble: 20 x 3
##                    country estimated.x estimated.y
##                      <chr>       <chr>       <chr>
##  1             Afghanistan         yes         yes
##  2                 Albania          no          no
##  3                 Algeria         yes         yes
##  4                  Angola          no          no
##  5               Argentina         yes         yes
##  6               Australia         yes         yes
##  7                 Austria          no          no
##  8                 Bahrain         yes         yes
##  9              Bangladesh          no          no
## 10                 Belgium         yes         yes
## 11                   Benin          no          no
## 12                 Bolivia          no          no
## 13 Bosnia and Herzegovina          no          no
## 14                Botswana          no          no
## 15                  Brazil          no          no
## 16                Bulgaria         yes         yes
## 17            Burkina Faso         yes         yes
## 18                 Burundi         yes         yes
## 19                Cambodia         yes         yes
## 20                Cameroon         yes         yes
```

So `left_join()`ing added a 7th column to our data, and if we have a look at it (`head()`), you can see that the yes's and no's from the estimated data have been repeated.

Great! Now we are going to continue with our analysis. We want to label our plots differently if they are based on estimated data. This means that when we go through our for loop, we want to conditionally plot them.

8.8 More R!

With just a little bit of time left, here are some things that you can look into more on your own.

8.8.1 Importing and Installing

Here are some really helpful packages for you to work with:

Remember you'll use `install.packages("package-name-in-quotes")` to install from CRAN.

- `readr` to read in .csv files
- `readxl` to read in Excel files
- `stringr` to work with strings
- `lubridate` to work with dates

You are also able to install packages directly with Github, using the `devtools` package. Then, instead of `install.packages()`, you'll use `devtools::install_github()`. And you can create *your own* packages when you're ready. Read http://r-pkgs.had.co.nz/ to learn how!

8.8.2 Organization and workflows

- set up a folder for figs, intermediate analyses, final outputs, figures

8.8.3 Getting help

You'll soon have questions that are outside the scope of this workshop, how do you find answers?

- end with a ton of resources: https://peerj.com/collections/50-practicaldatascistats/

8.9 Ideas for Extended Analysis 2

- stringr() http://r4ds.had.co.nz/strings.html

Chapter 9

Collaborate with GitHub

9.1 Overview

The collaborative power of GitHub and RStudio is really game changing. So far we've been collaborating with our most important collaborator: ourselves. But, we are lucky that in science we have so many other collaborators, so let's learn how to use GitHub with one of them.

We are going to teach you the simplest way to collaborate with someone, which is for both of you to have administration privileges. GitHub is built for software developer teams, and there is a lot of features that we as scientists won't use immediately.

We will do this all with a partner, and you'll work pretty independently, with us providing help as you go.

Objectives

- create a new repo and give permission to a collaborator
- open as a new RStudio project!
- collaborate with a partner, explore github.com blame, history
- practice more ggplot2 collab

1. add their neighbor as a collaborator to their repo
2. practice more; make changes to their repo, and to their neighbor's.

Resources

9.2 create gh-pages repo and give someone permission

9.3 clone to a new Rproject

9.4 Rmd analysis together

- Rmd analysis