Introduction to Open Data Science

Julie Lowndes & The OHI Team

2017-11-21

Contents

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 6.1: Overview and Prerequisites.

Suggested breakdown for a 2-day workshop:

time	Day 1	Day 2
9-10:30	Motivation, R and RStudio	Wrangling (tidyr)
break		
11-12:30	GitHub, scripting in R	Extended analysis (importing, for loops, etc)
lunch		
13:30-15:00	Visualization (gpplot2)	Collaborating with GitHub
break		
15:30-17:00	Wrangling (dplyr)	Extended analysis 2

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.

2.1.1 Tidy data workflow

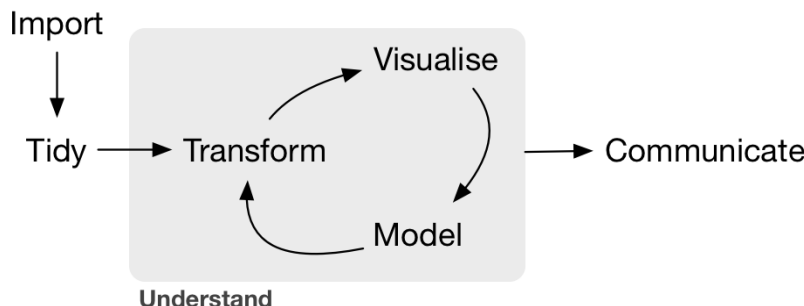We will be learning about tidy data.

Figure 2.1:

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 Orientation, GitHub Setup

3.1 Objectives & Resources

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.
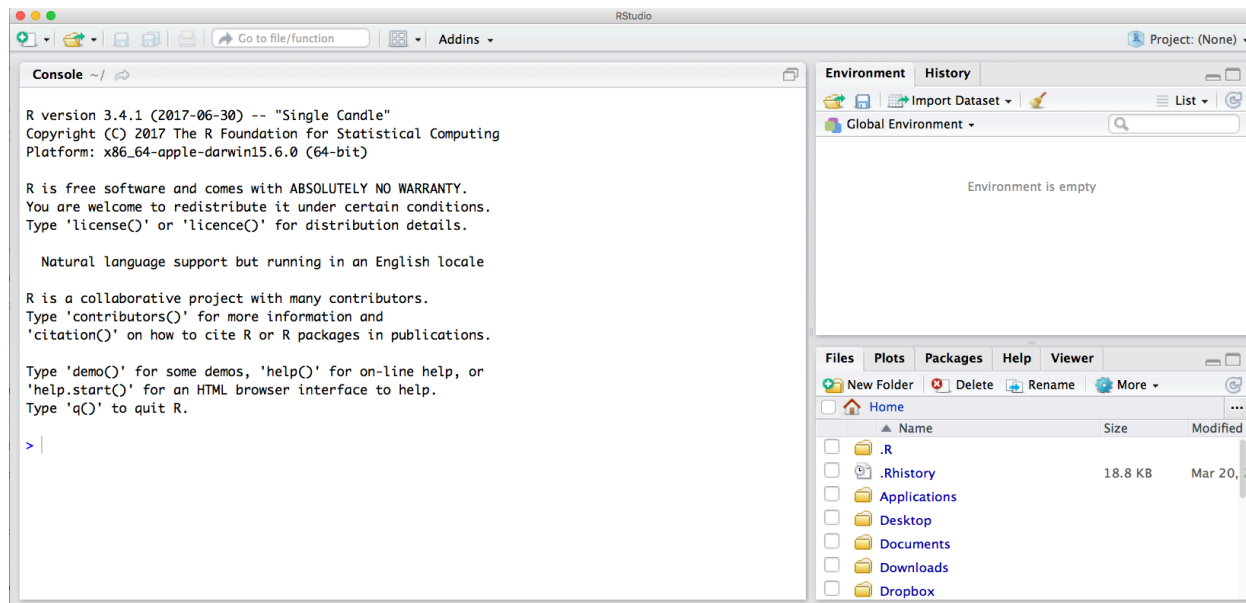
Figure 3.1:

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 (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)
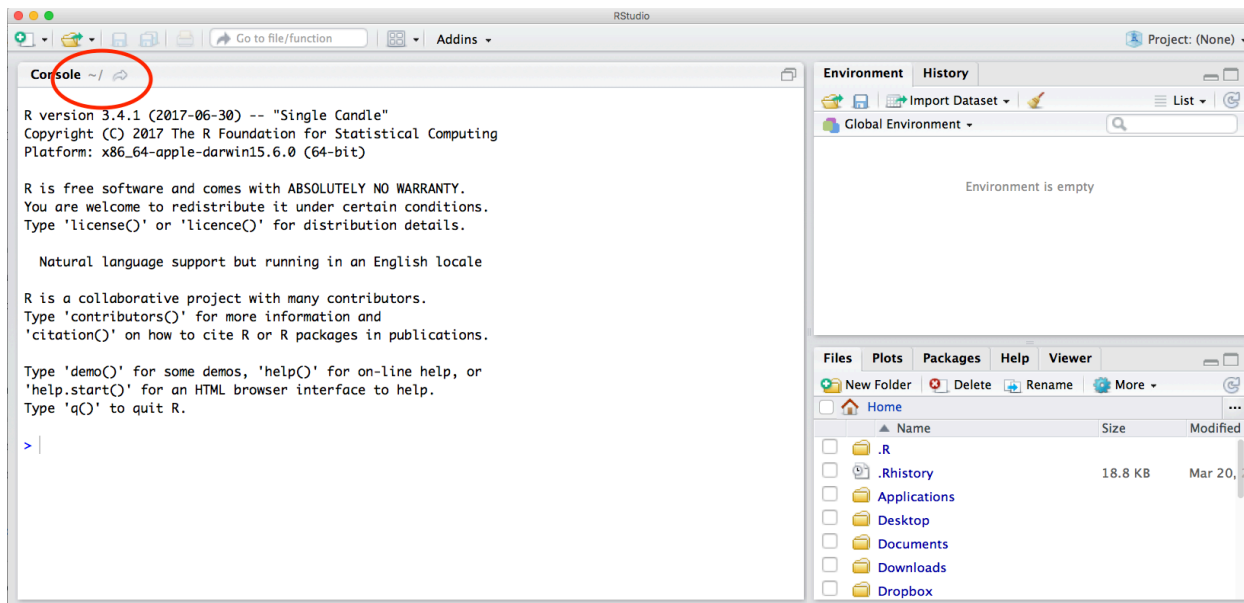- Environment/History (tabbed in upper right)

Figure 3.2:

- Files/Plots/Packages/Help (tabbed in lower right)

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

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

If you've 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.
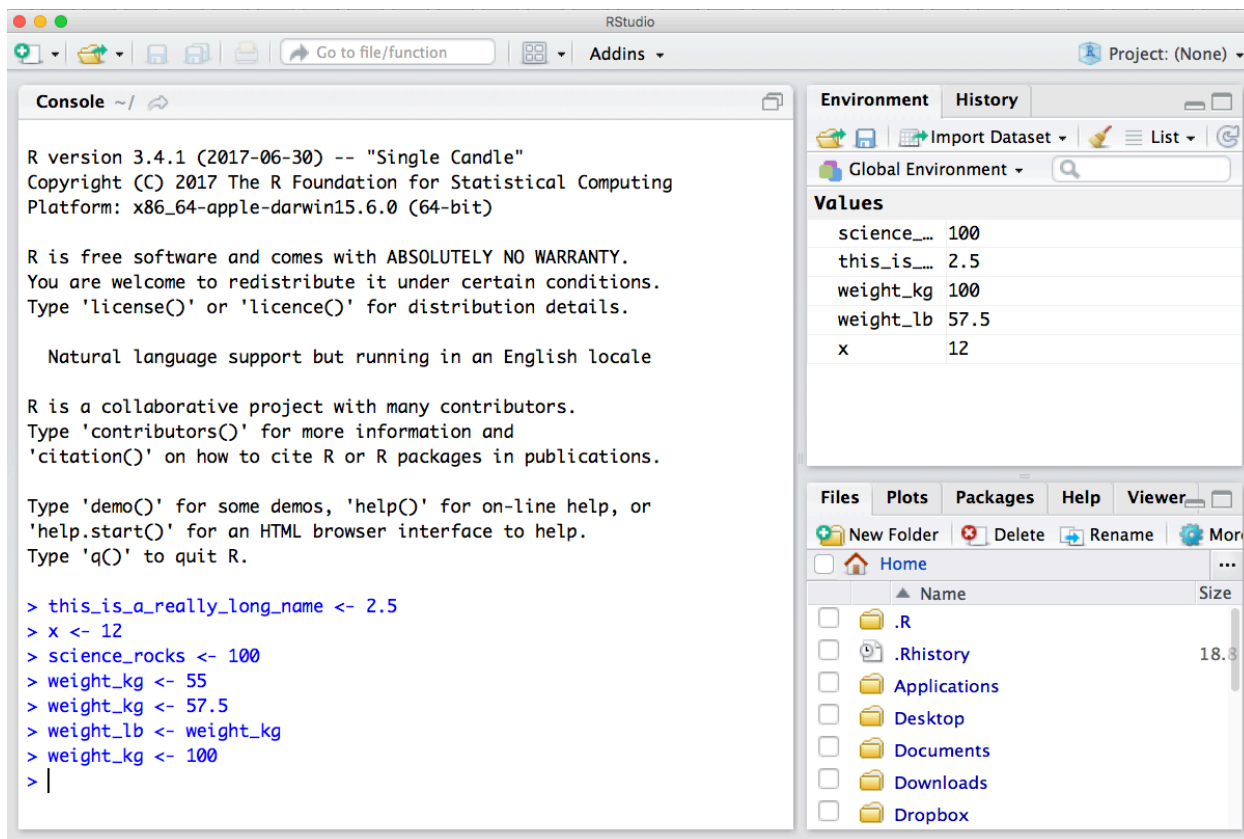
Figure 3.3:

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] "Tue Nov 21 15:54:39 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"
```

```r
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:

```r
rm(weight_kg)
```

To remove everything:

```r
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 differnt 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:

```
1 ▾ ---
2   title: "My Project"
3   author: "Julie"
4   date: "11/21/2017"
5   output: html_document
6   ---
7
8 ▾ ```{r setup, include=FALSE}
9   knitr::opts_chunk$set(echo = TRUE)
10  ```
11
12 ▾ ## R Markdown
13
14  This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word
    documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.
15
16  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:
17
18 ▾ ```{r cars}
19  summary(cars)
20  ```
21
22 ▾ ## Including Plots
23
24  You can also embed plots, for example:
25
26 ▾ ```{r pressure, echo=FALSE}
27  plot(pressure)
28  ```
29
30  Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that
    generated the plot.
31
```

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:
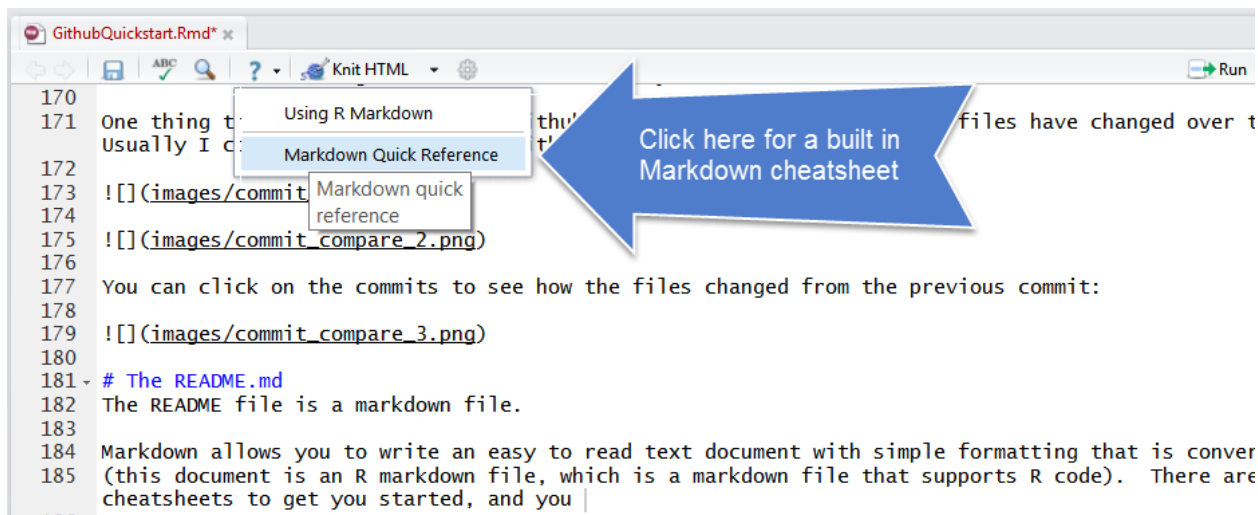
See Mastering Markdown · GitHub Guides and add some more personalized content to the README of your own, like a bulleted list or blockquote.

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

3.6.1   Your Turn

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

3.7   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.

3.8 Troubleshooting

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

3.8.1 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
+
```

3.8.2 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

Using RStudio+GitHub, and R Scripts

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. create an R script
6. practice the RStudio-GitHub workflow by editing and adding files

4.1.0.2 Resources

These materials borrow from:
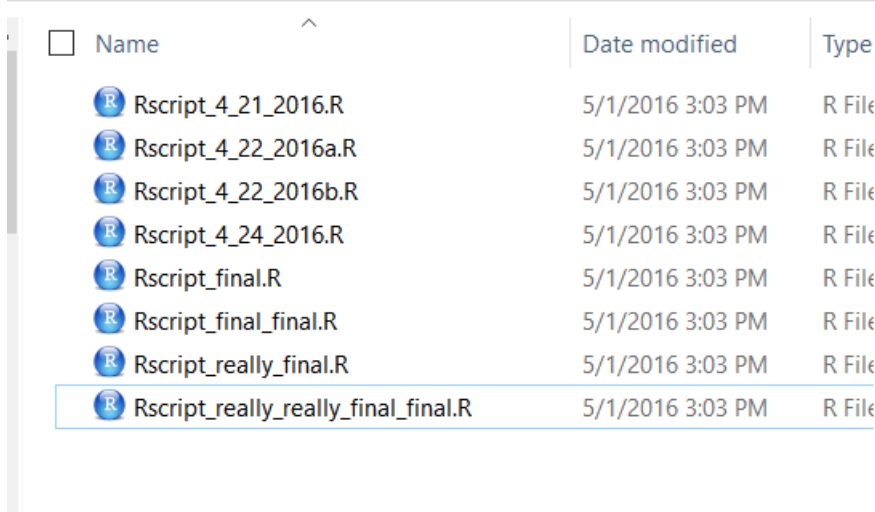
- Jenny Bryan's lectures from STAT545 at UBC: The Shell
- Jenny Bryan's Happy git with R tutorial
- Melanie Frazier's GitHub Quickstart

  • 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
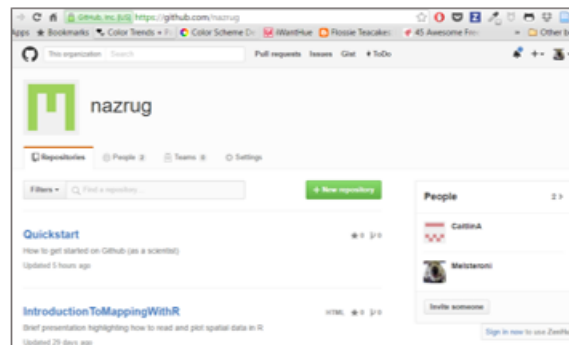
4.3   Create a repository on Github.com
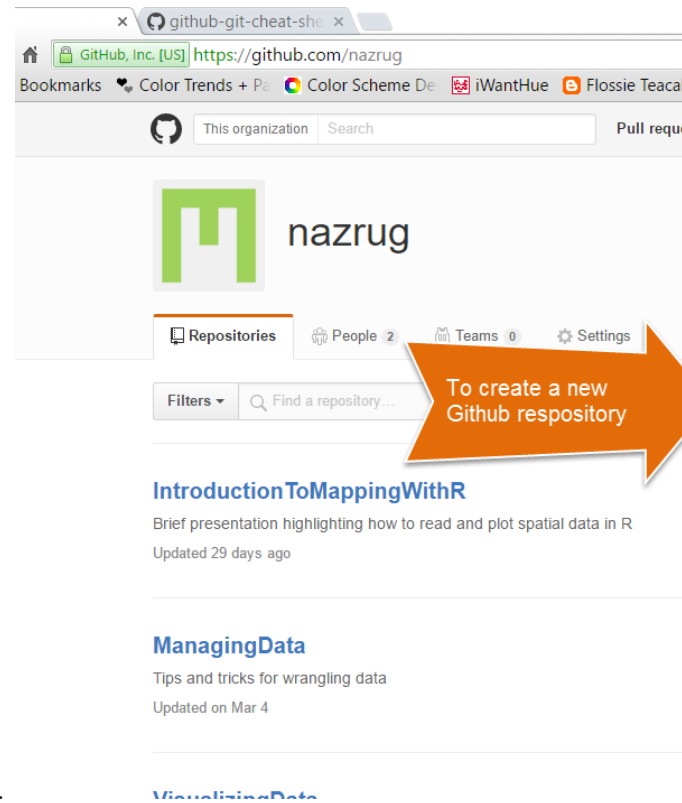
First, go to your account on github.com and click "New repository".

Choose a name.Call it whatever you want (the shorter the better), or follow me for convenience. I will call mine `my-repo`.
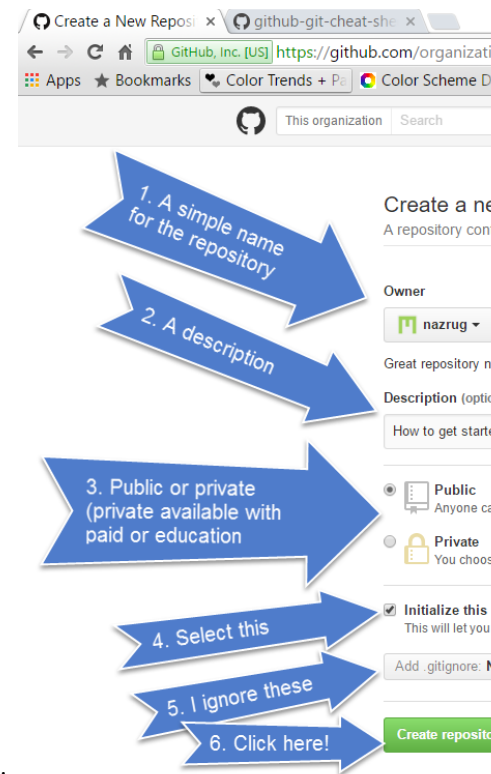
Also, add a description, make it public, create a README file, and create your repo!

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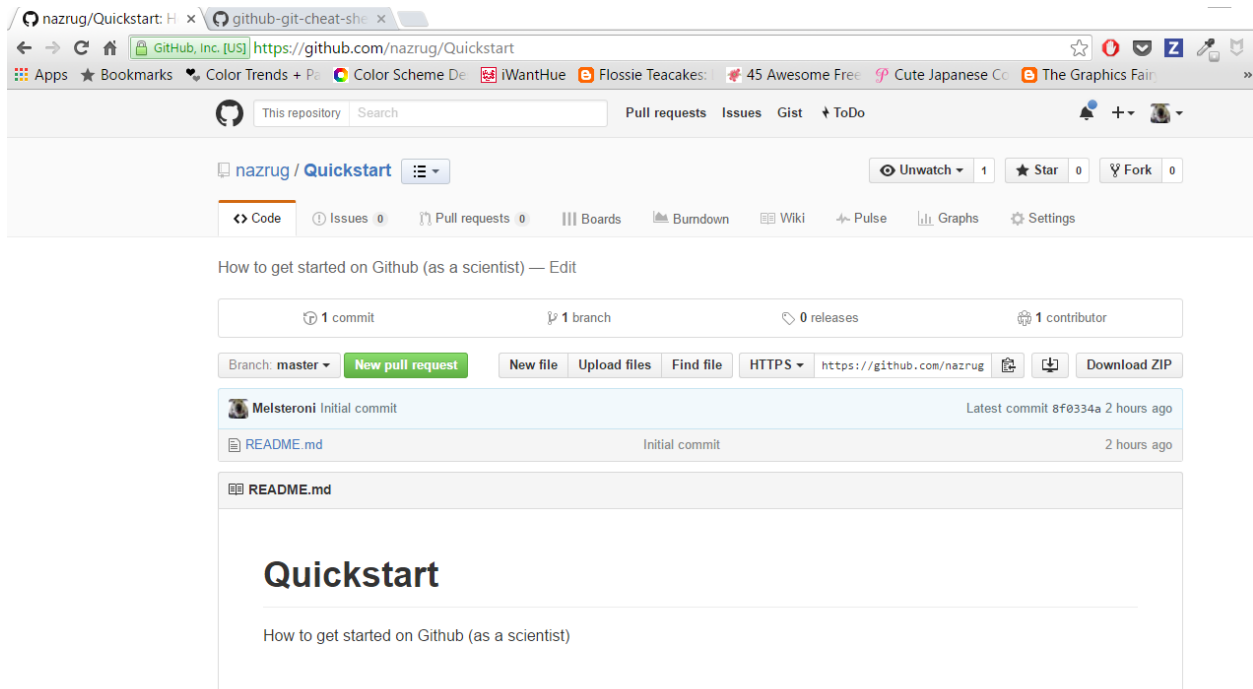
From here, you will work locally (on your computer).

4.4   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).
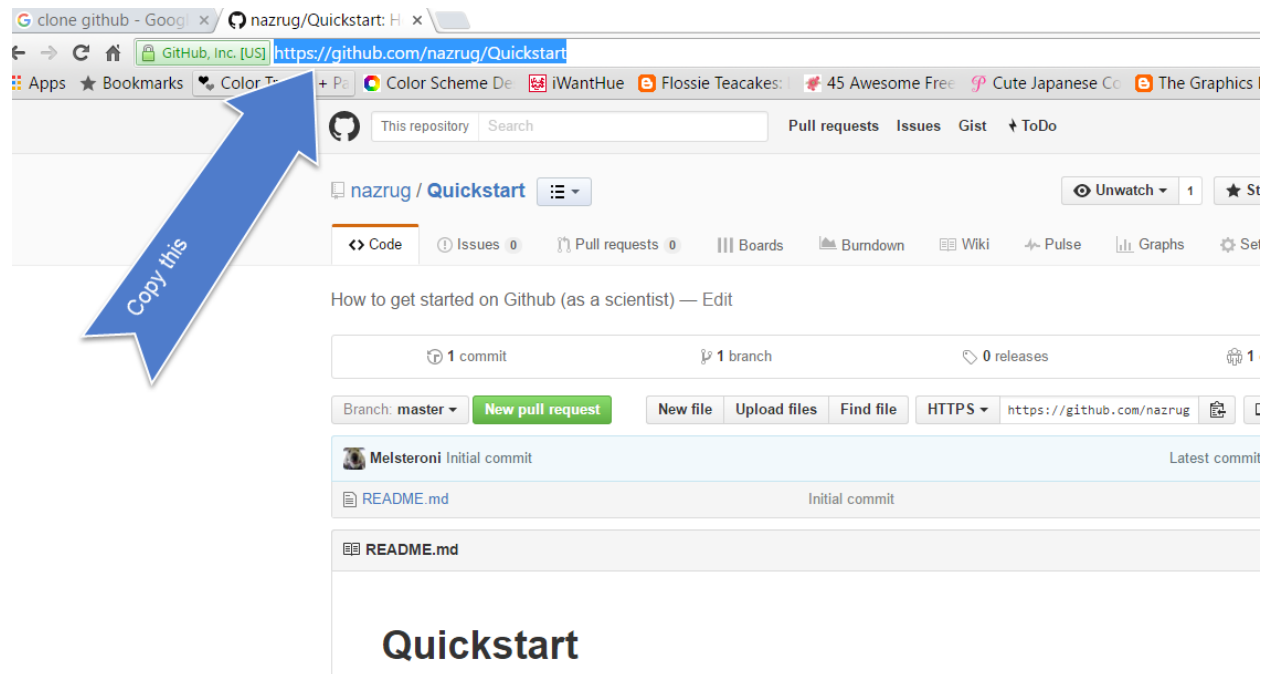
Figure 4.1:
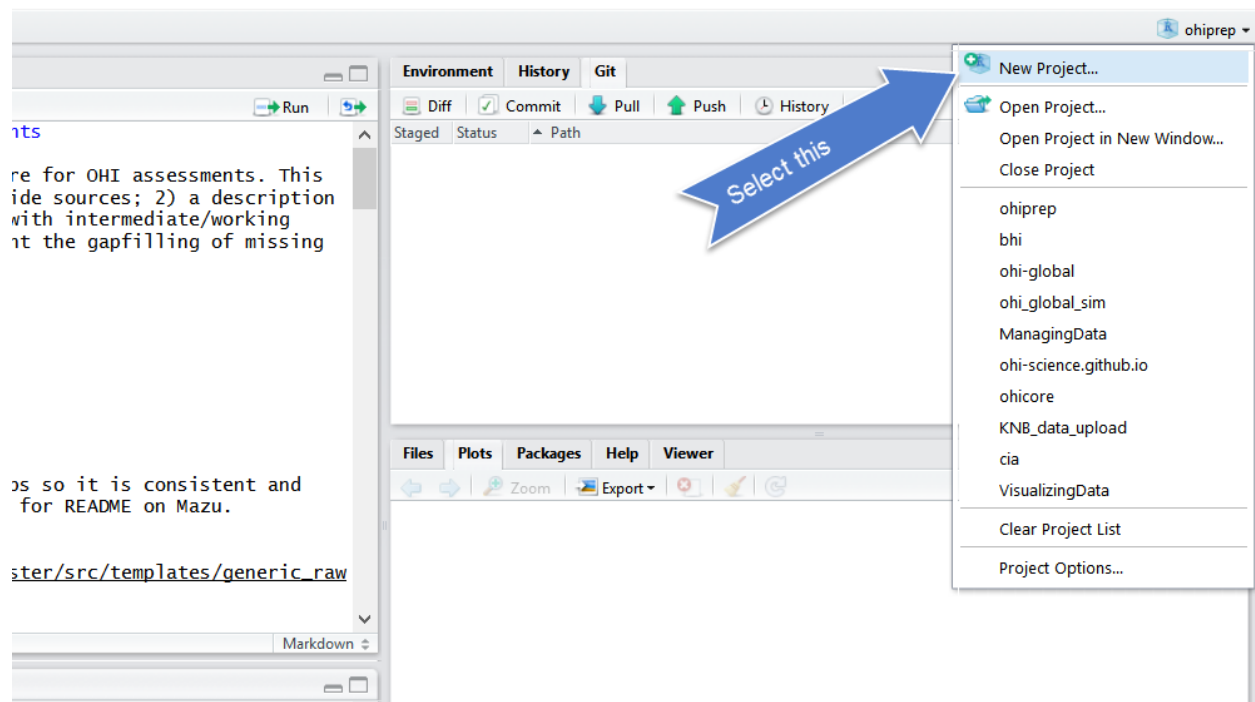
Step 3: Select Version Control
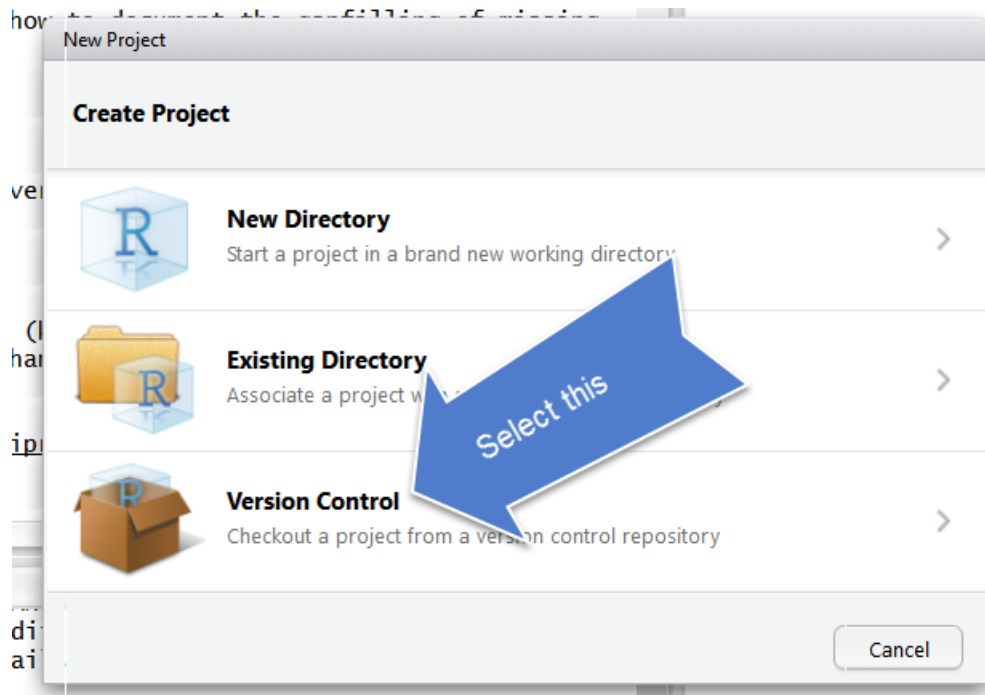
Step 4: Select Git
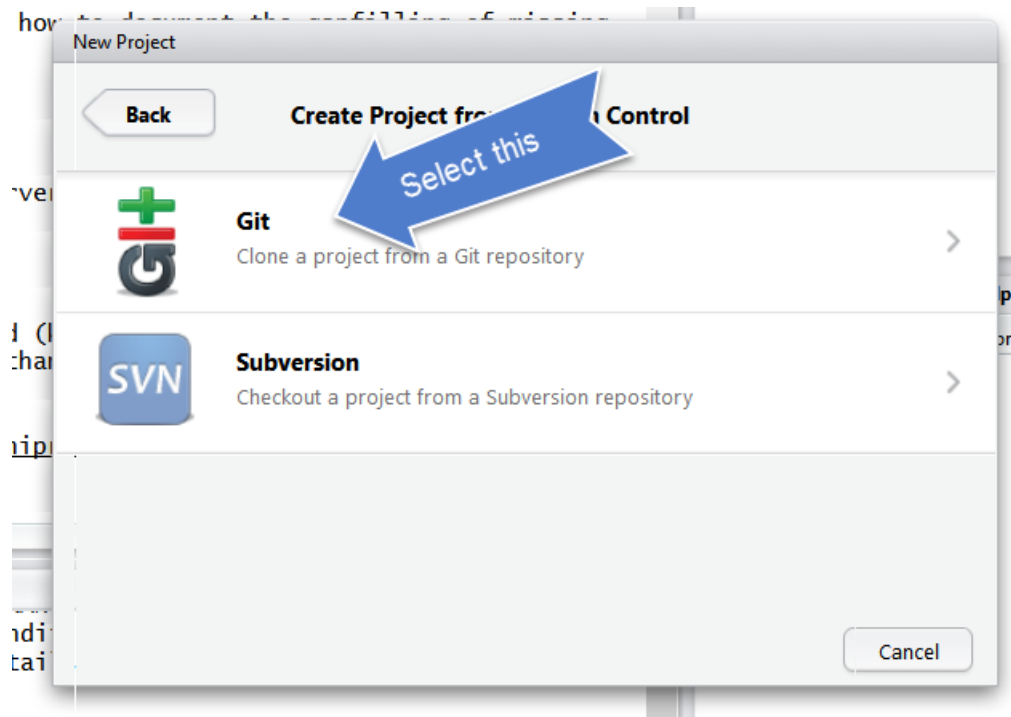
Step 5: Paste it in the Repository URL field, and type tab to autofill the Project Directory name. Make sure you keep the Project Directory Name THE SAME as the repository name from the URL.

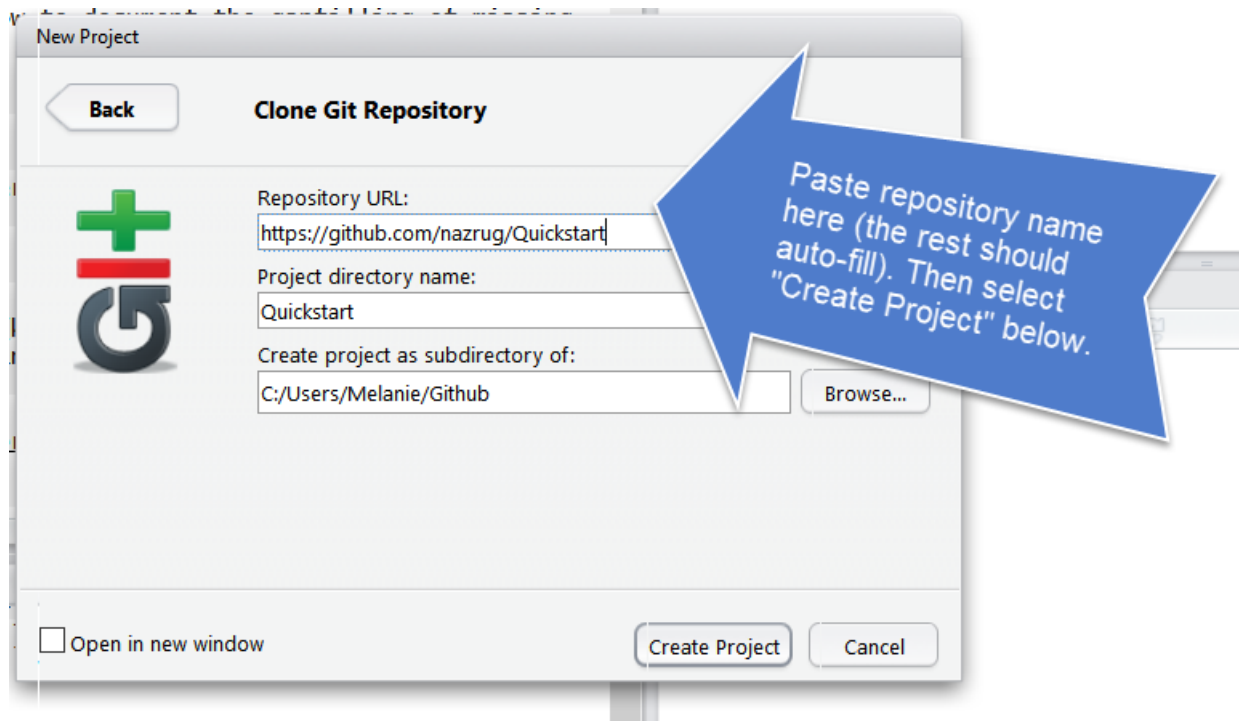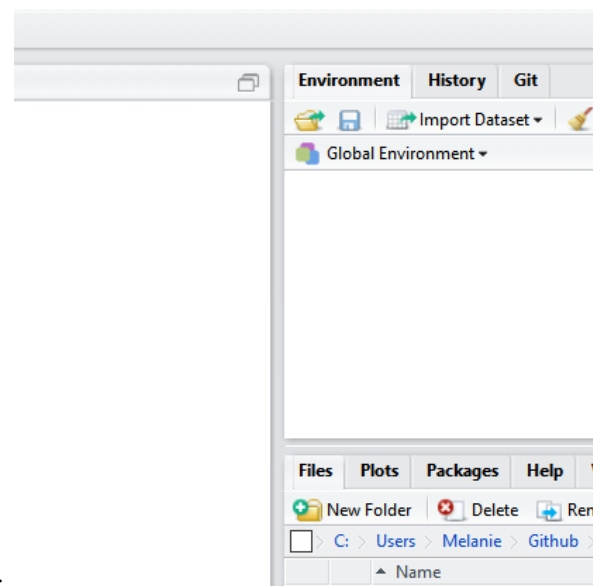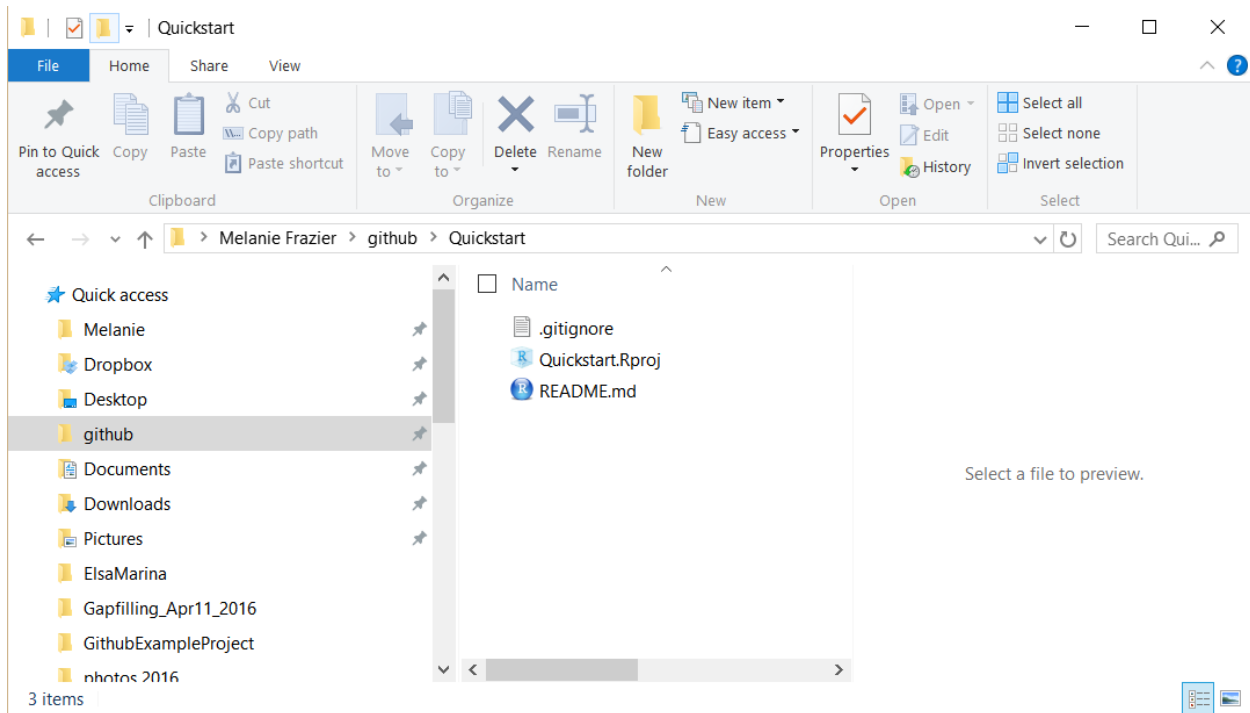Save it in your github folder (click on Browse) to do this.

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.

4.5   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

4.6   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.

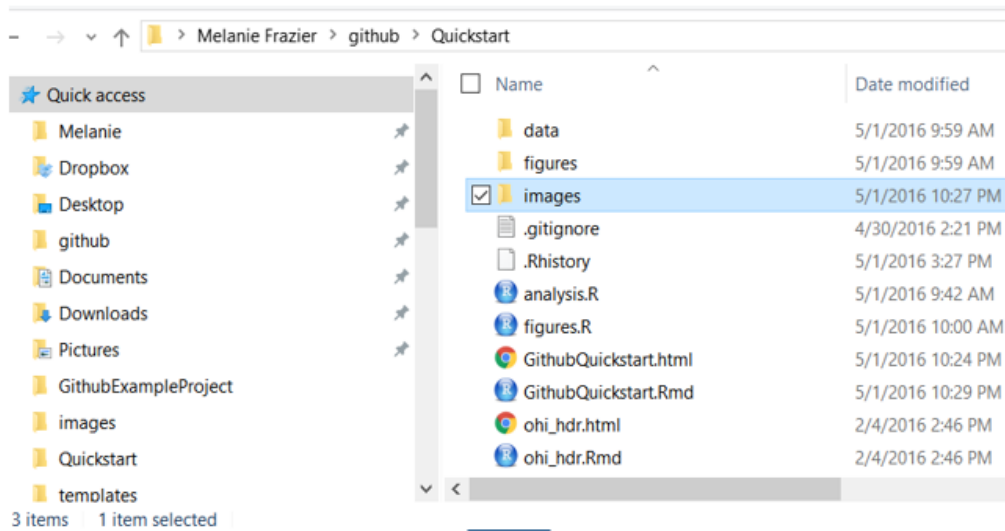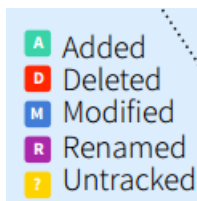To make changes to the repository, you will work from your computer ("local Github").
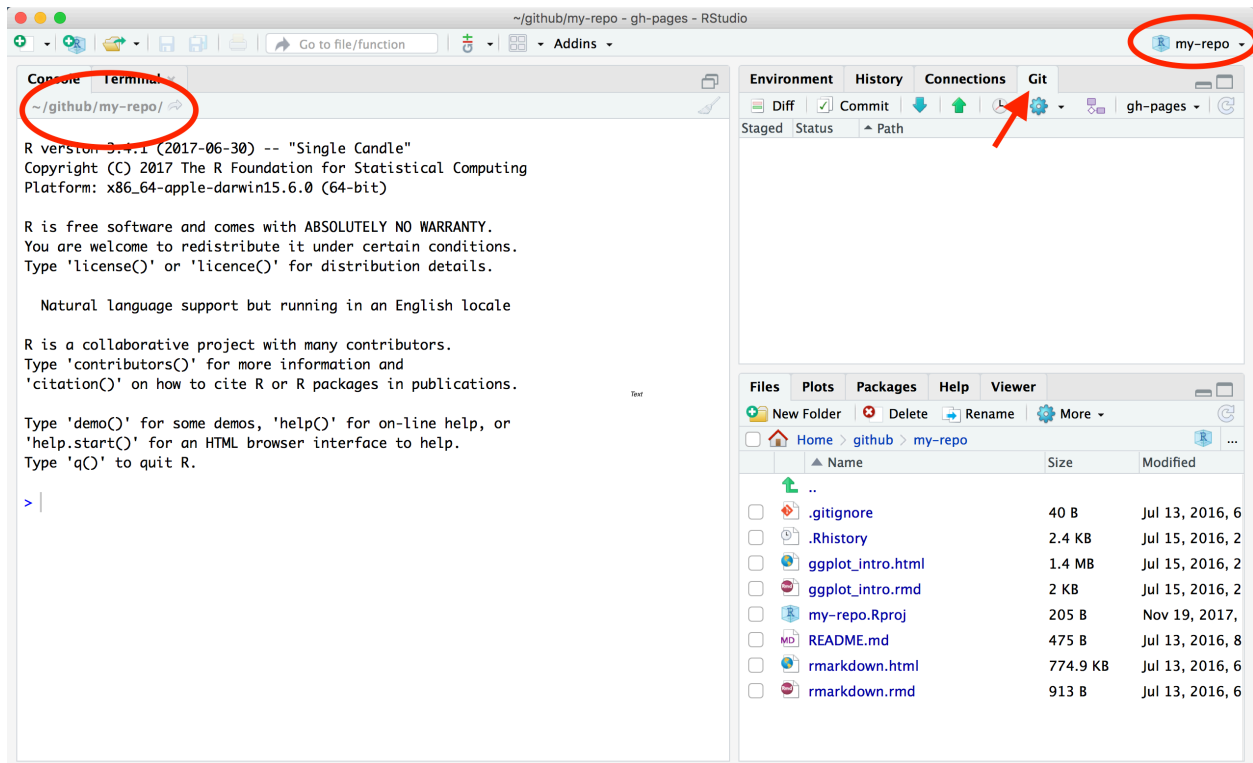
Figure 4.3:

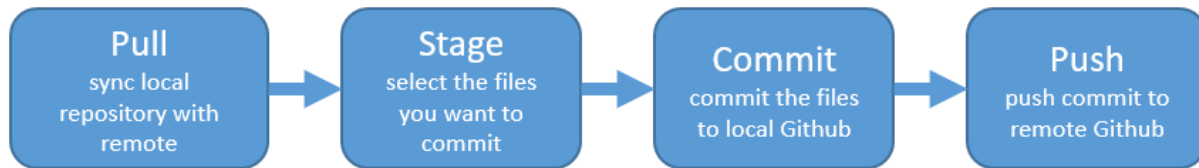When files are changed in the local repository, these changes will be reflected in the Git tab of RStudio:
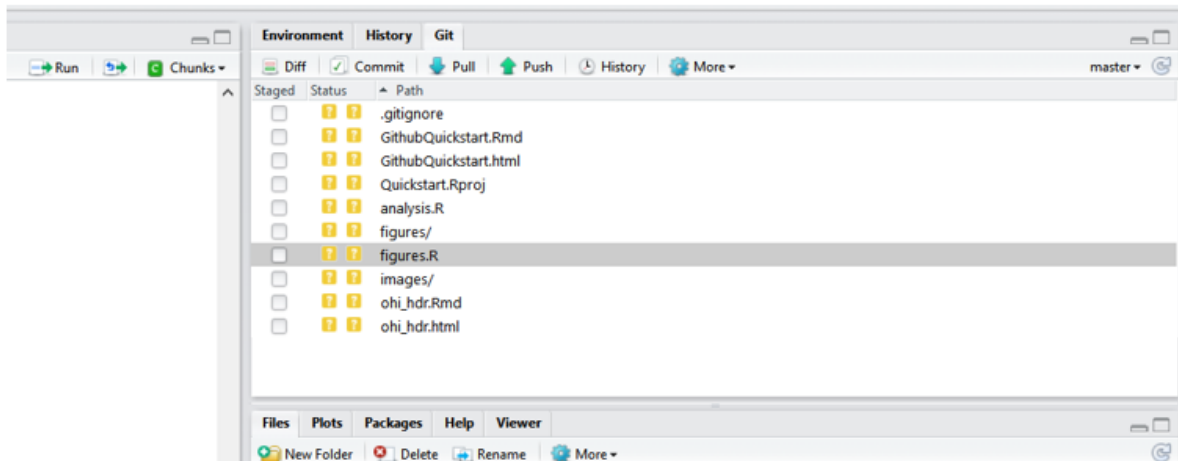
4.6.1   Inspect what has changed

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

4.7   Sync from RStudio to GitHub

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


We walk through this process below:

4.7.1   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.
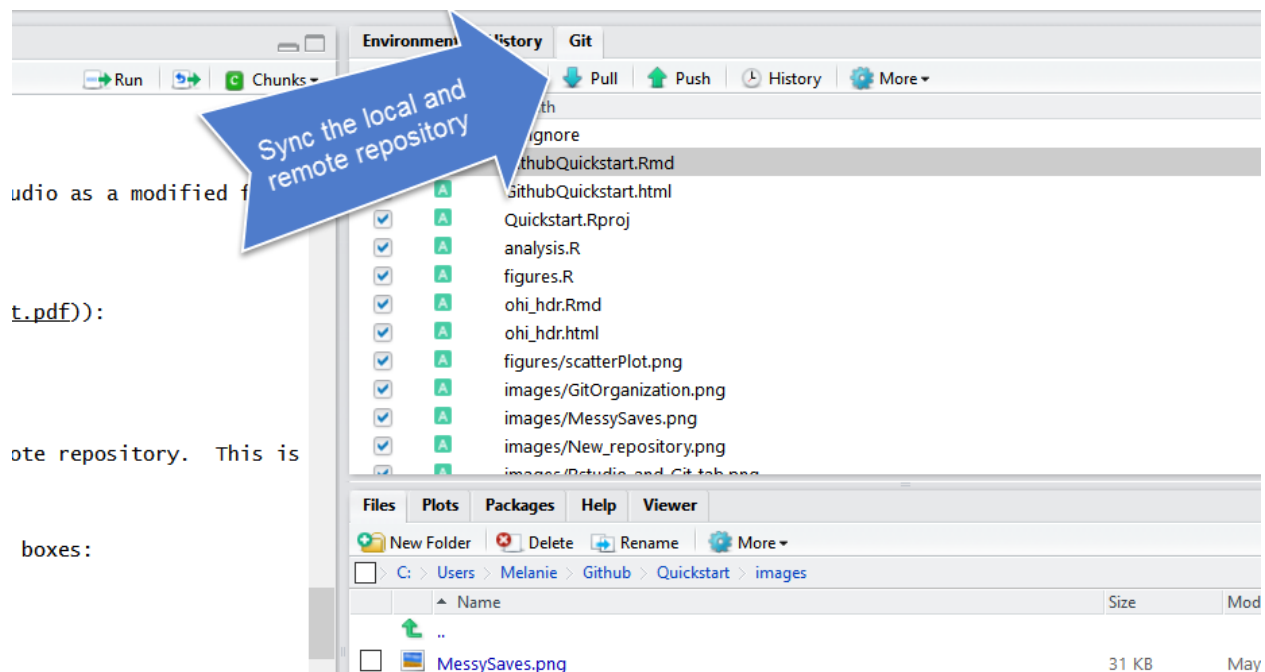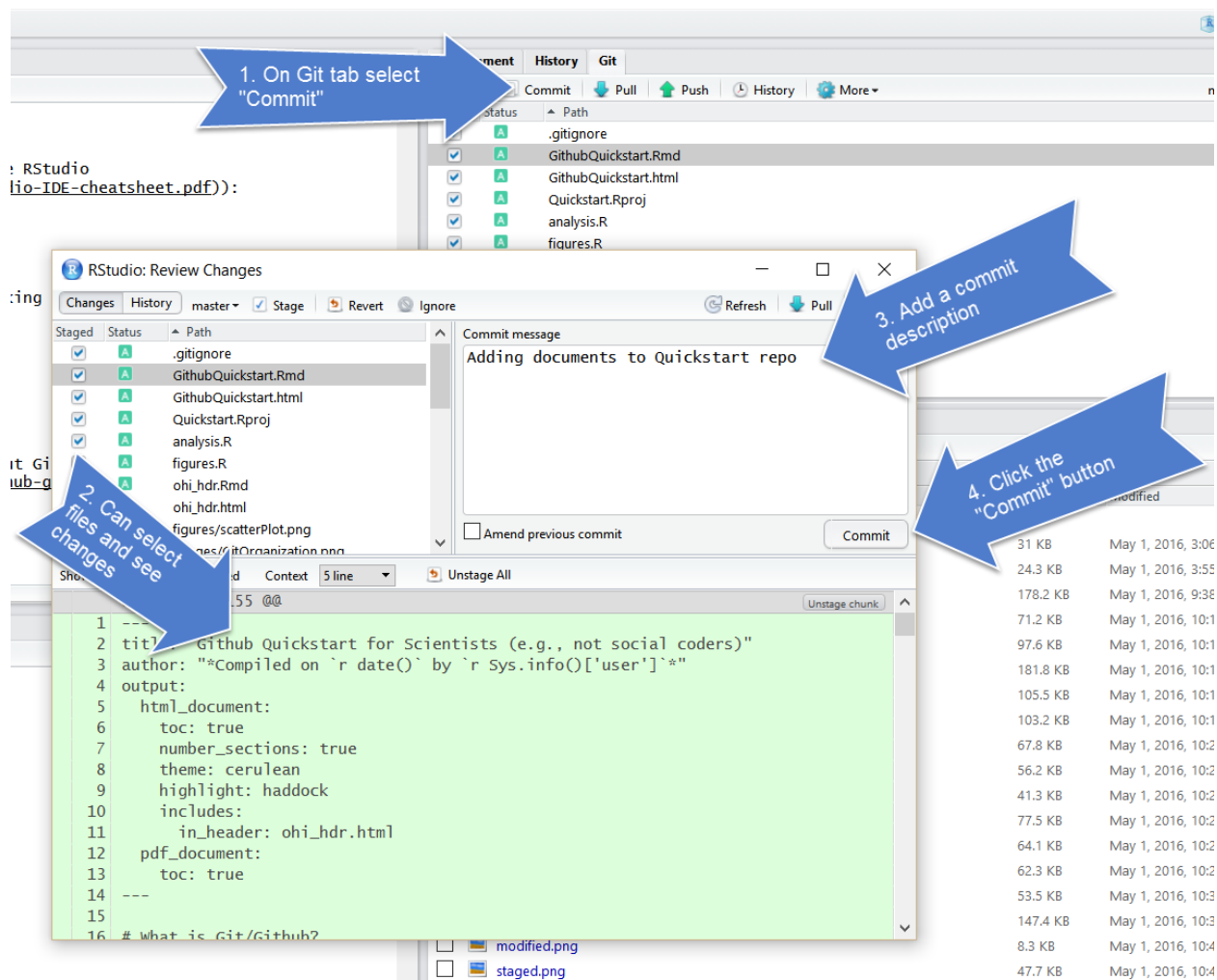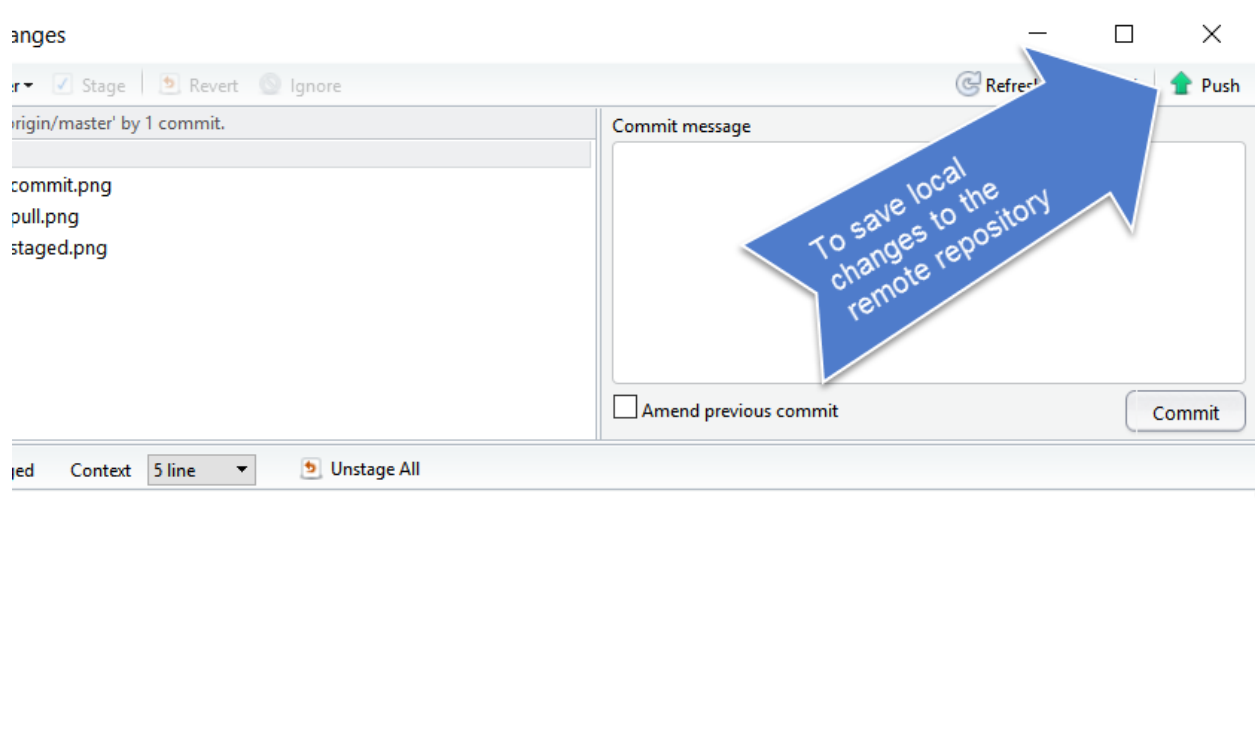

4.7.2   Stage

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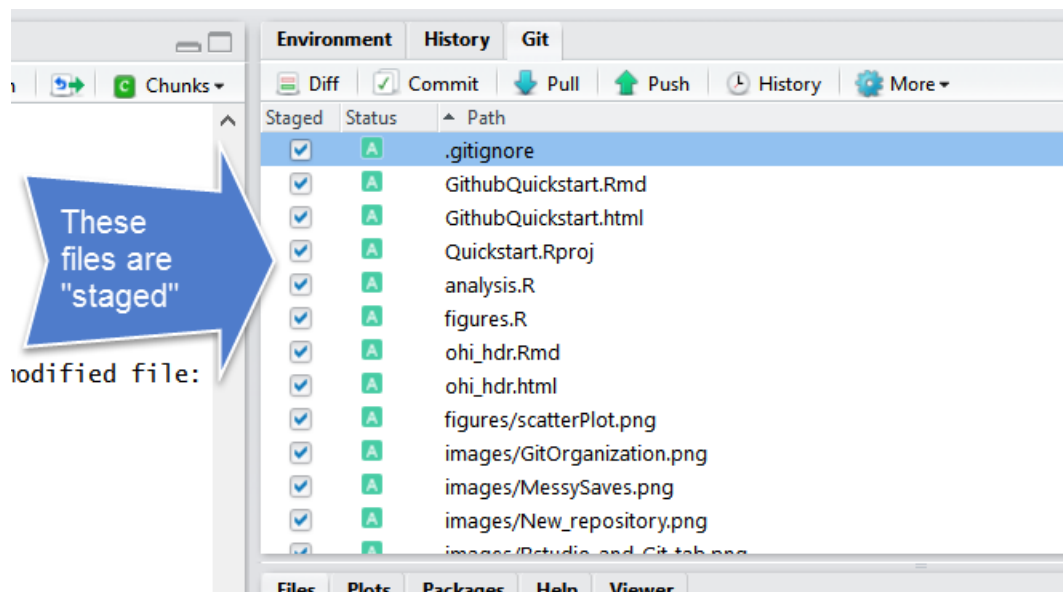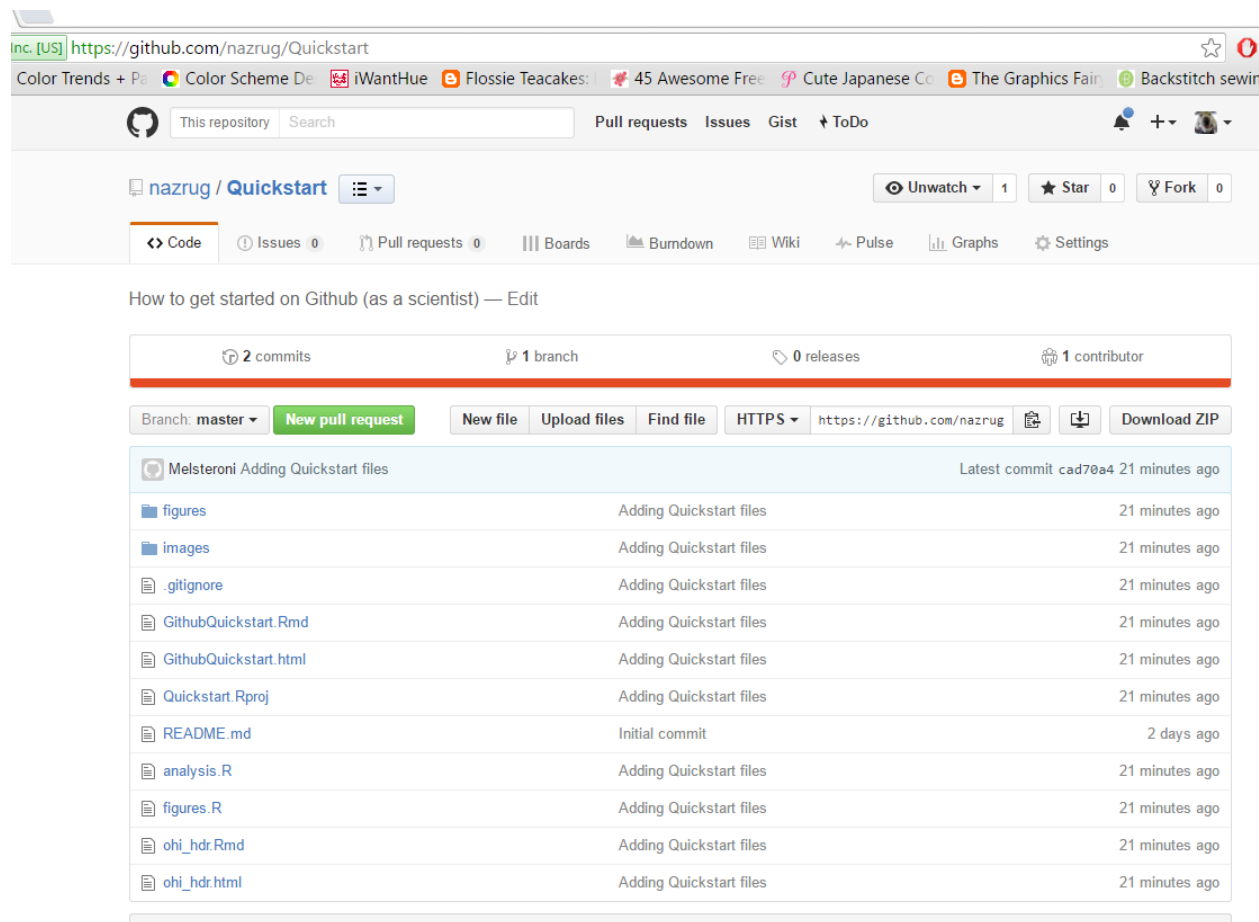
4.7.3   Commit

4.7.4 Push

4.8 Explore remote Github

The files you added should be on github.com:

Let's also explore commit history, file history.

4.8.1 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!

4.9 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 script 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 script (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

4.9.1   Your turn

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

Then, sync your script to GitHub.

4.10   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:

4.11   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).

You can follow along this morning with the Desktop App: 3-minute youtube video

Chapter 5

Visualizing data

In development.

5.1 Objectives & Resources

5.1.0.1 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 (maybe call it mpg_viz.R?)
- practice rstudio-github workflow
- use and credit http://r4ds.had.co.nz/data-visualisation.html

Why do we start with data viz? 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

- Conceptual and building, cheatsheet images

- http://r4ds.had.co.nz/data-visualisation.html

- https://pdfs.semanticscholar.org/d779/6f85dabccd18673f382c100fc06f55e8b501.pdf

5.1.0.2 Resources

Here are some resources that helped make this tutorial: - R for Data Science - ggplot2-cheatsheet-2.0.pdf - Interactive Plots and Maps - Environmental Informatics - Graphs with ggplot2 - Cookbook for R - ggplot2 Essentials - STHDA

D Robinson: - http://varianceexplained.org/r/why-I-use-ggplot2/; add screenshot of ggplot2 cheatsheet - http://varianceexplained.org/RData/

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 included in 'base R'; they are 'out-of-the-box' functions. You can also install packages from online 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/.

```r
## 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 my 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   Aesthetic mappings

5.4   push to GitHub

5.4.1   Your turn

5.5   Common problems

...

Basics

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same few components: a **data** set, a set of **geoms**—visual marks that represent data points, and a **coordinate system.**
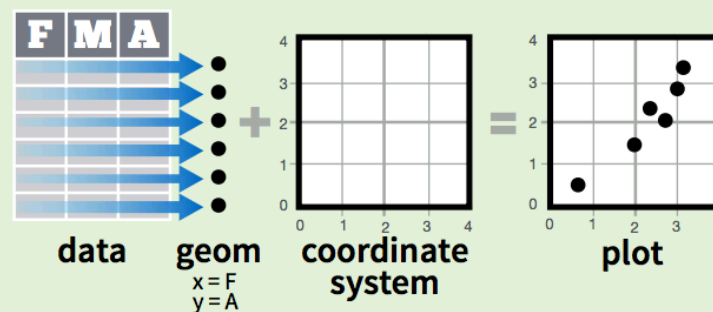
To display data values, map variables in the data set to aesthetic properties of the geom like **size**, **color**, and **x** and **y** locations.
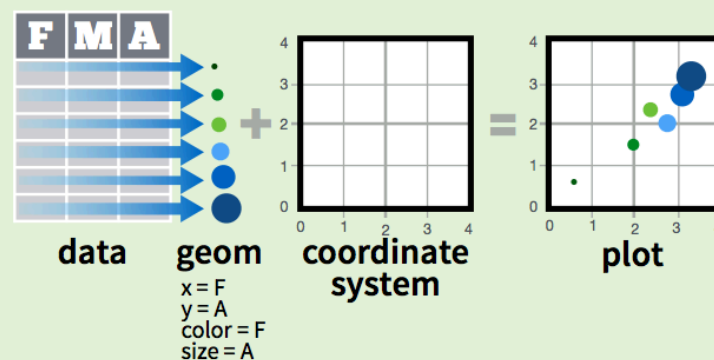
Figure 5.1:

5.5.1   Your turn

5.6   The Layered Grammar of Graphics

5.7   push to GitHub

Explore GitHub; show that your fig has a URL you could share with someone

5.8   end of the day: intro to Markdown (README)?

so they can post a figure on a readme?

5.9   Troubleshooting

Chapter 6

Wrangling (dplyr)

6.1 Overview

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')
```
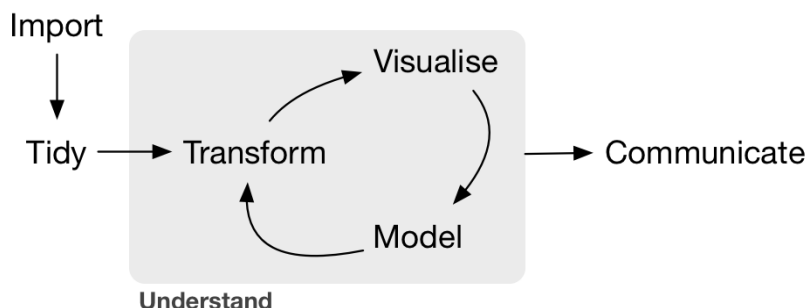
Figure 6.1:

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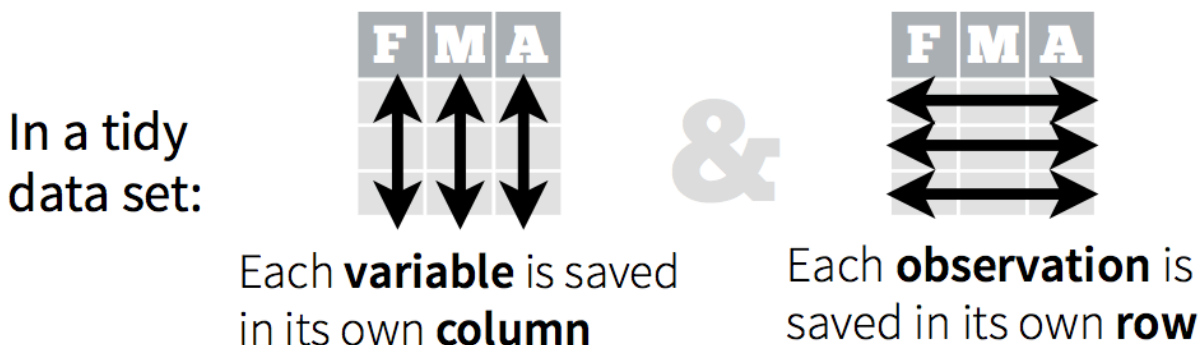
The `mpg` dataset we were working with this morning was an example of tidy data. When data are tidy, you are

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:
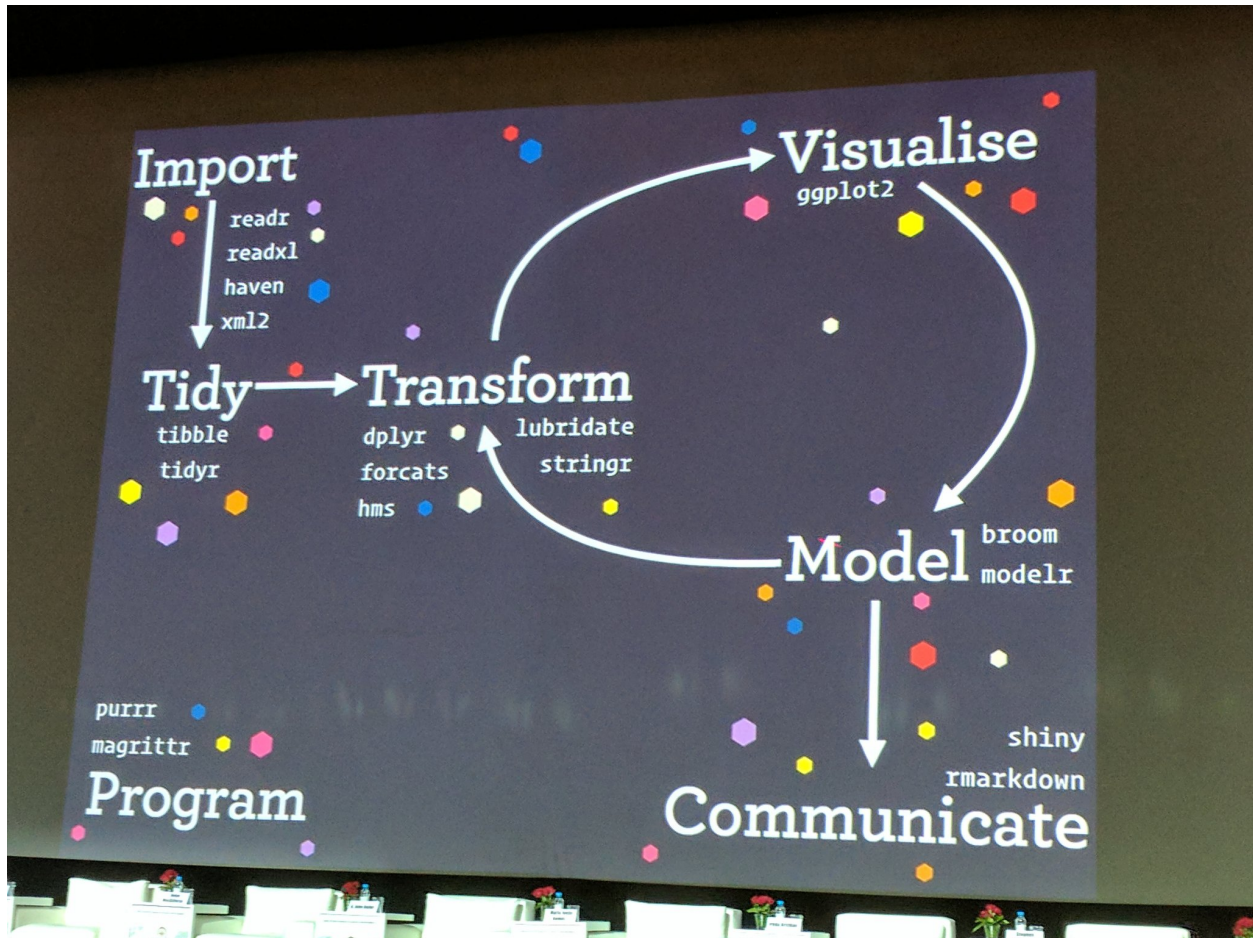
Figure 6.2:

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:

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

6.4   Explore the gapminder data.frame

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

The data are on GitHub, in our course webite. 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!). And for comparison, you could also do it in base-R.

```
## 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/
```

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` overview

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:

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

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.

6.5.1 install `tidyverse` (which has `dplyr` inside)

In your R Markdow 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).

6.6 Use `dplyr::filter()` to subset 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):

Subset Observations (Rows)

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 expenctancy 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).

Subset Variables (Columns)

Figure 6.4:

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, <-. Similarly, 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.

6.9   Use `dplyr::select()` to subset 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.

Make New Variables

Figure 6.5:

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
```

6.10 Use `dplyr::mutate()` to add 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)
```

6.11 Your turn

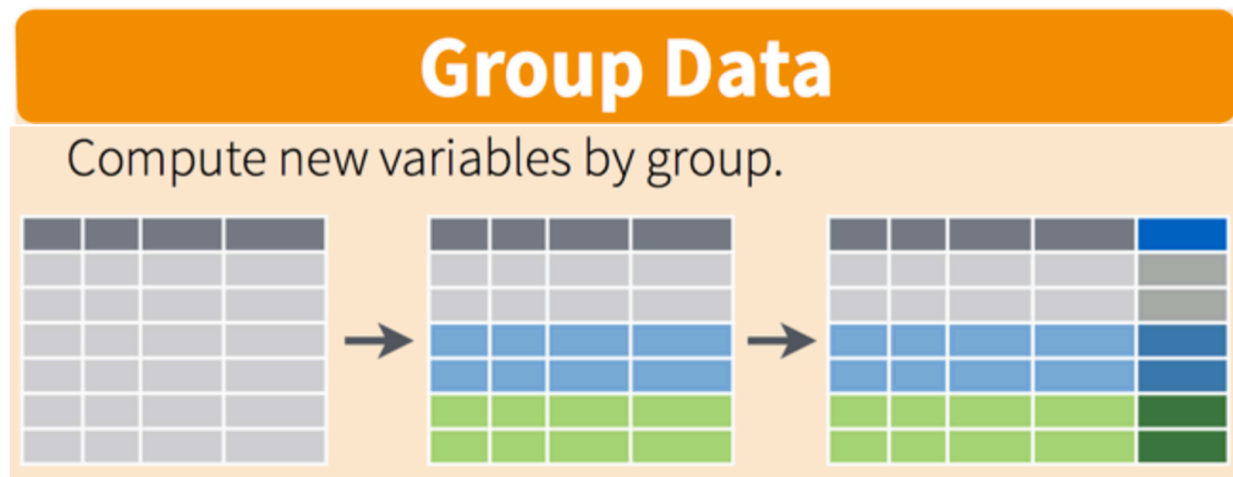Find the maximum gdpPercap of Egypt and Vietnam Create a new column.

Figure 6.6:

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

6.11.1   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") %>%
  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 gdp-Percap/max_gdpPercap for each year or something. But maybe you just wanted that max_gdpPercap for something else. Let's keep going...

6.12   `dplyr::group_by()` to operate 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:

6.12.1 `dplyr::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).

6.13 `dplyr::arrange()` to order

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)
```

6.14 Your turn

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

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

Do the following:

1. arrange your data frame in descending order (opposite of what we've done). Expect that this is possible: use the Help pages.
2. save your data frame as a variable
3. Knit your RMarkdown file
4. Sync your .Rmd and .html to GitHub (pull, stage, commit, push)

6.14.1 Answer

…

6.15 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.