==========
==========

Welcome to Software Carpentry Etherpad for the January 10-11th. workshop at the University of Connecticut!

This pad is synchronized as you type, so that everyone viewing this page sees the same text. This allows you to collaborate seamlessly on documents.

Use of this service is restricted to members of the Software Carpentry and Data Carpentry community; this is not for general purpose use (for that, try https://etherpad.wikimedia.org/).
Users are expected to follow our code of conduct: http://software-carpentry.org/conduct.html
All content is publicly available under the Creative Commons Attribution License: https://creativecommons.org/licenses/by/4.0/
We will use this Etherpad during the workshop for chatting, asking questions, taking notes collaboratively, and sharing URLs or bits of code.

Website: https://mickley.github.io/2019-01-10-UCONN/

Socrative Login: https://b.socrative.com/login/student/
Room: UCONNSWC1

==========
==========

Instructors:

* Tim Moore (EEB) - timothy.e.moore@uconn.edu
* James Mickley (EEB) - james.mickley@uconn.edu
* Cera Fisher (EEB) - cera.fisher@uconn.edu
* Dyana Louyakis (MCB)
*Jeremy Teitelbaum (MATH)

Helpers:

* Eliza Grames
* Pariksheet Nanda (MCB) - pariksheet.nanda@uconn.edu
* Dipanjana Dalui
* Alex Trouern-Trend (EEB) - alexander.trouern-trend@uconn.edu

Attendees: (Put your name & dept here):

* junfeng tian Geography
* Lidia Beka (Molecular and Cell Biology)
*Ahmad Hassan (Molecular and Cell Biology)
Janeth Perez Garza (Animal Science)
*Andrew Stillman (Ecology and Evolutionary Biology)
* Mark Stukel (Ecology and Evolutionary Biology)
* Sarah Peck (Computer Science and Engineering) - I took notes in a different format that may be helpful. You can find them on my git: https://github.uconn.edu/smp07012/RinGit/blob/master/R_Basics.R!
* Megan Chiovaro (Psychological Sciences - Ecological)
* Bernard Goffinet (EEB)
* Ji Won (Geography)
Amanda Hewes (Ecology and Evolutionary Biology)
*Jamie Micciulla (Molecular and Cell Biology)

*Tanner Matson (Ecology and Evolutionary Biology)
*Justine Liu (Animal Science)
*Ben Ranelli (EEB)
*Shu Jiang(Psychological Sciences)
*Adam Rainear (Communication)
Sinead Sinnott (Clinical Psychology)
Kurt Schwenk (Ecology & Evolutionary Biology)
Ian Sands (Biomedical Engineering)
Meghan Maciejewski (EEB)
Jie Chen (Nursing)
Xiaowen Liu (Education)
Christian J. Connors (Ecology & Evolutionary Biology)
*Shengyun Gu (Linguistics)
Eileen Schaub (Ecology & Evolutionary Biology)
*Saurav Dhar (Computer Science and Engineering)
Jessica Lodwick (Ecology and Evolutionary Biology)
Sara Horwitz (Ecology and Evoluntioary Biolgy & The Children's Museum)
Pariksheet Nanda (Molecular and Cell Biology)
Rishabh Kejriwal (Molecular and Cell Biology)
Chloe Jones (Psychological Sciences)

Nearby options for lunch:

==========

- Cafe Coop, UConn Coop
- The Student Union
- Bookworms Cafe, Homer Babbidge Library
- The Benton Cafe

==========
==========

Day 1 - R

Setup:

1. Sign in up front: Get a name tag and a sticky note of each color
2. Download and install software from our course website: **https://mickley.github.io/2019-01-10-UCONN/**
3. Put your name and department under Attendees above (You can get here from the Etherpad link on the course website)
4. Fill out the pre-workshop survey if you haven't already: **https://www.surveymonkey.com/r/swc_pre_workshop_v1?workshop_id=2019-01-10-UCONN**
5. **Enter your name at the top of the chat column of this etherpad (top right corner)**
6. Download the datasets from the course website and save/unzip them to your desktop (see GAPMINDER DATA below)
7. In a web browser sign in to your GitHub.com account
8. In RStudio click on Tools > Global Options... > Git/SVN > Make sure the grey box under "SSH RSA Key" shows a file name and a "View public key" link exists next to it and clicking on "View public key" opens a window opens which is not blank. If the "SSH RSA Key" box is blank, click "Create RSA key..." and leave the optional Passphrase blank and click "Create".

TO DO:

1. **Take a name tag and sign in up front**
2. **Grab 2 stickie note (1 of each colour)**
3. **Install R, Rstudio, and Git**
4. **Create Github account**
5. **Introduce yourself here in the etherpad (above). You can also enter your name in the chat (top right)**

*****Please put up a blue stickie on your laptop if you have signed up for Github and have all three programs installed (R and Rstudio)*****

SOCRATIVE LINK: https://b.socrative.com/login/student

Student/ ROOM CODE: UCONNSWC1

- **TIM's DROPBOX LINK for gapminder data:** https://www.dropbox.com/sh/ae86tlimhtoq48v/AAARAcJ2yGxSE523_q0mV9R3a?dl=0

==========

DAY 1 - Introduction to R and RSTUDIO

- Increasing font size in RStudio:
    - Someone had a question this morning about the RStudio font being too small on their high resolution laptop screen.
    - You can fix this from **Tools > Global Options ... > Appearance > Zoom** My fonts look more reasonable at 130%
- Using an R script is recommended for everything you do. This really helps with reproducibility. Also, you will likely forget how you did something 6 months from now.
- Tim's RScript is synced on his dropbox, so if at any point you want to get Tim's current work, you can open it at his dropbox: https://www.dropbox.com/sh/ae86tlimhtoq48v/AAARAcJ2yGxSE523_q0mV9R3a?dl=0
- (but if you're lost, put your red sticky up!)
- R can be used to carry out operations, like addition, subtration
    - and follows order of operations
- You need to save the script as a .R file to activate the syntax warnings (Nice catch, Cera!)+1
- Using Command-Enter (or Ctrl-enter) on a line in the editor window sends it to the console (like clicking run)
- Commenting
    - Use a # at the beginning of a line. Everything after a # is never run by R
    - I comment nearly every line of code I write. Try to say why you're doing something, rather than just what it is
    - Command Enter on Mac runs a line of code; Control Enter on PC runs a line of code
- Storing something in a variable
    - Use <- to store something to a variable
        - You can also use =, but this is less preferable in modern R (e.g., x = 4)

        - Values are displayed in console at upper-left corner
        - variable names are case-sensitive
        - Variable name cannot start with number, period, or underscore
        - Check your Global Environment to make sure that your variable name is stored
        - Good variable names are descriptive, but don't require a lot of typing. E.g., having a dataset named My_SouthAfricaData_part2_version12 (replaced) will be a lot of work to type each time you use it
        - Avoid variable names that serve as functions (ex. min)
        - #if you want to run the entire block of code, then you could highlight all the lines and run it
- To remove things from the global environment
    - In console type:
    - rm(list=ls())
    - **this will remove all values that you previously assigned EVEN if they were saved into your R script

In R, the 'assignment operator' is <- : in many languages it is "=" but R wants to distinguish testing for equality from assigning a value to a variable.

If you try to use a variable before giving it a value, you will get an error. For example, if you haven't used y before, and you do **x<-y** you will get an error because R doesn't know what y is and it wants to give the value of **y** to **x**

Q) Why do we call things variables in R instead of objects?
A) Because unlike most other programming languages R distinguishes between variables and functions. For example you can have both a variable named "min" without affecting the functon "min()":

```
min <- 3
min(1:5)  # surprisingly, still works!
# min is variable of type numeric vector:
class(min)
# delete the variable
rm(min)
class(min)
```

Also see Hadley Wickham's guide explaining objects in R: http://adv-r.had.co.nz/OO-essentials.html

- Vectors
    - A list of values of the same data type
    - E.g. 1:5 = 1 2 3 4 5
    - If you do math on or with a vector, it does it to all the values at once
        - e.g. (1:4)^2

Some important **functions** (for cleaning up):

-
    - ls() gives a list of all objects stored in environment.
    - rm() removes objects from environment, use "list" parameter for removing a set of objects.

Packages -

- A **package** is a set of functions that extend the functionality of R to solve particular problems. For example, a package might implement a fancy statistical test that isn't included in the basic language R; by including that package you gain the additional functions. The package **ggplot2**, for example, greatly extends the plotting functions available in R.
    - Can be installed in the command line using the install.packages() command.
    - Can be installed from the RStudio GUI using the "Install" button under the Packages tab.
    - Can also be loaded by checking box in system library under the Packages tab.
    - To install a package: install.packages("ggplot2") # note the quotes are required
    - To install multiple packages at once: install.packages(c(" ", " ", ...))
    - To load a package to use it: library(ggplot2) # Note that quotes are not required
        - NOTE: when you use library() to load a program, you may get a warning message about certain functions or objects inside that package over-riding another function of the same name belonging to a previously loaded package
            - For example, library(dplyr) showed this warning: The following objects are masked from ‘package:stats’: filter, lag
            - Be aware that if you need filter from package:stats as you work, it will no longer be available. Sometimes this can be the source of an error when you work in R
            - Need to be aware of package functions that have been "masked"

Reproducible work in R-

- Best Practices: http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745
- Good Enough Practices: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005510
- Use RStudio projects to organize your work.
    - Treat your raw data as sacred and never change it
- How to Find Your Project Folder to add data- More --> Show Folder in New Folder

Getting Help

- Getting help in R for a function
    - ?log # Get help on the log function by putting a question mark before it
    - help(log) # Another way to get help for a function
    - ??read.ta # Two question marks search for a keyword in R help, searches all packages
    - If you want more information about a specific package:
    - vignette("package.name")
    - More websites for help:
        - https://cran.r-project.org/
        - https://stackoverflow.com/
        - When searching in stackoverflow, use the tag [r] in the search bar
        - To help with reproducibility when asking questions
            - use the sessionInfo() function, lists all your packages, their versions, and R version
            - use dput to package up your workspace to a file

- One stackoverflow comment: nearly every question has already been asked, so googling around often gets you an answer more efficiently than actually asking a new question.

Dataframes

- In R, we work a lot with dataframes, sort of like a spreadsheet, with rows and columns
- to make a dataframe use the dataframe() function
- Sidenote: R commands can take up multiple lines, e.g., with the dataframe example
    - cats <- data.frame(coat = c("calico", "black", "tabby"),
    - weight = c(2.1, 5.0, 3.2),
    - likes_string = c(1, 0, 1))
- The 'c' command creates a vector -- the c stands for 'combine' (or maybe concatenate)
- Working with dataframes
    - To get a particular column, use a $ after the dataframe name followed by column name
        - e.g., cats$weight
    - To examine the contents of a dataframe
        - Just type the name and run it
        - view(cats)
        - str(cats)
            - # The structure function also gives us the number of rows and columns, and tells us what type of data is in each column
                - Factors" are special ways to store categorical data
        - nrow() and ncol() give the number of rows or columns
        - dim() gives the number of rows and columns
        - colnames() gives the names of the columns of a dataset

Working with Vectors

- You can combine vectors of the same type of data (numeric, character) using the c() function
- You can do arithmetic with numeric vectors: num_vector + num_vector2
- The seq() function can make sequences of numbers
- You can name elements in a vector by using the names() fuction
    - Note that you are storing TO a function, its on the left side of the <- assignment operator
    - names(my_example) <- c("a", "b", "c", "d")
- **LETTERS** is a vector of upper case letters
- **letters** is a vector of lower case letters

Loading Data into R

- When you run data("gapminder") it's normal to see it in the environment as grayed out and saying "<Promise>" instead of a "data.frame". If you try to see the access gapminder by typing gapminder and running it, you will see it evaluate into a data frame.
- Most of the time, we are loading our data in. Best to have it in CSV format and use read.csv:
    - read.csv("data/gapminder_data.csv") # Note the filename/folder has to be in quotes

Subsetting data in a vector or dataframe (the basic R way)

- use square brackets

- x[1] # First element
- x[1:4] # First four elements
- x[c(1, 3)] # First and third elements
- For datasets, you put two numbers inside the square brackets, row first, then column
    - gapminder[1, 2] # The data in the 1st row, second column of the dataframe

- The first element in your vector is"1" not "0"
    -
    - Referred to as "1-based indexing"
- When referencing rows or columns using square brackets, the number on the left side of the comma is the row and the number on the right side of the comma is column.

#----------#

Day 1 - Afternoon Session

#----------#

GIT Lesson Resources
socrative room: TEITELBAUM4084
instructor shell history: http://teitelbaum.ngrok.io
Socrative Login: https://b.socrative.com/login/student/

Mac People: Open Terminal
Windows People: Open Git>Git bash

Git allows users to track progress over time through a project, and to set check points.
Git is an incredibly powerful version control system, designed for open source development

Goals:
1. How to track progress, set benchmarks and reset changes.
2. Learning how to use Git to collaborate with multiple users.

It is mainly designed to be used at the Command line.

Note: the "$" symbol is the prompt. It's okay if your prompt is different and longer.

How to survive on the command line:

1. pwd - print working directory - find where you are in the file heirarchy. If you get lost, type pwd
2. ls - list - show contents of your current directory.
    2.1. use ls -a to show all
3. cd - change directory
4. mkdir - make a new directory

To personalize Git

- git config --global user.name "Firstname Lastname" # Replace with your own name
- git config --global --user.email "your.email@uconn.edu" # Replace with your own email
- git config --global core.autocrlf input # Mac version
- git config --global core.autocrlf true # windows version
- git config --global core.editor "nano -w"
- git config --list # to show what your configuration settings are

Git commands

- **git init**
  - This creates a folder .git/ for tracking the folde
- **git status**
  - Ask git what's going on
  - This is a very useful and frequently used command. We often want to know what is tracked

Create a Text file

-
  - nano mars.txt
  - add text: "Mars looks red from a distance."
  - in nano use cntrl [^]X to exit, and say yes to save changes
  - Use cat command to see contents of created file (if you need to get out of cat is use cntrl[^] d)

Levels of Git

- Work Area = where work is happening, where files you're working on are being modified
- Staging = intermediate area for staging files before adding to a repository
  - use **git add** mars.txt
    - This adds files from the work area to the staging area
- Repository + where we want to store out files, so that we can keep track of this version
  - use **git commit (important)**
    - This saves files from the staging area to the repository
    - add the flag '-m' to avoid text editor opening and asking for a message (e.g. git commit -m "Hello world!")

Note Branches can be "branched" to maintain parallel versions of repositories. We won't be covering this.

Seeing changes in a file

- **git diff**
  - Lines with +++ in front are added, lines with --- in front were removed
  - Only shows changes between staging area and working area
  - To show differences between repository and staging, use **git diff --cached**
    - Often, this is written as **git diff --staged** as well

Seeing changes in a repository

- use **git log**
  - to get an idea of what changes you have made to a repository -
    - these can get really, really long
    - If you see ":" and not "END" - it means there is another page for which "spacebar" will take to the next page and "Q" will get you out of it
    - **git log -1** will the most recent change, git log -2 will give the two most recent ones, and so on
    - **git log --oneline**
      - Show an abbreviated version of the list of commits
    - space bar will page and Q will quit.

"." is current directory. You can add a directory to a staging area which will add all of the files in that directory at once to the staging area.

Going back in time with git:

- to discard recent changes of a file and go back to the version in the repository
    - **git checkout -- mars.txt**
        - Caution: This overwrites the file in your work area
- The latest version of all files on the repository is called **HEAD**
    - The 2nd most recent version is **HEAD~1**, 3rd most recent **HEAD~2**, etc. (note that these are **tildas**, not dashes)
    - e.g. compare latest version to the one before it:
        - **git diff HEAD~1 HEAD**
    - To just show the change
        - **git show HEAD~2**

Interacting with your cloud repository on Github:

-
    - From the command line:
        - **git push origin master** send changes to remote server
        - **git pull origin master** take files from remote server to your local repository
        - **git clone** Take a repository from github and store a copy on your machine
            - You only do this if you're the collaborator
            - e.g., **git clone https://github.com/jeremy9959/swc.git collaborator**

Dealing with conflicts:

-
    - From the command line, you pull down changes from the repository to your machine:
- git pull origin master
-
    - Git will alert you that there are conflicting changes between your version of the repository and the remote version
    - To resolve this, you will have to visually inspect the files. Git will mark the files to show you where the conflicting lines are,

and you will decide which lines to keep and which to discard by editing the file. Then do:
git add <conflicted file>
git commit <conflicted file> -m "resolved conflicts"
git push origin master ## to send the resolved repository back to the remote version.

Licensing:

- Read about licensing
    - https://www.fsf.org/licensing
    - https://creativecommons.org/
- You can add a license to an existing github project
    - https://help.github.com/articles/adding-a-license-to-a-repository/

#----------#

Day 2 - Morning Session

#----------#

##########
URGENT:

DID ANYONE FIND A WALLET IN THIS ROOM YESTERDAY?

TALK TO TIM
##########

Welcome back!

Website: https://mickley.github.io/2019-01-10-UCONN/
Socrative Room name: **LOUYAKIS**

TO DO:

1. **Take a name tag and sign in up front**
2. **Grab 2 stickie note (1 of each colour)**
3. **Install R, Rstudio, and Git if you haven't already**
4. **Install ggplot2, dplyr, tidyr, knitr**

Link to code for ggplot2:
https://www.dropbox.com/sh/3mi1m73qmcxotmy/AAAALSy_FaXPsKCygigrg7cqa?dl=0

How to check version of your package in R:
> packageVersion("ggplot2")

The base command for making a plot is **ggplot**

aes (x,y)= Aesthetic

We have to specify the type of plot we want by following the ggplot2 function with a '+'

and use **geom_point()** to make a scatterplot
we can add a 'color' aesthetic to the aes to change the color of the points by a variable in your data

We can use **geom_line()** to change our plot to a line plot

**** You can think of ggplot as taking on layers:**

- the base layer is the geom
- you can add various layers to your plots using '+' and different **geom functions** (e.g., geom_line, geom_point)
- Help on geometry layers: https://ggplot2.tidyverse.org/reference/#section-layer-geoms
- Common geometry layers:
    - **geom_point()** # Scatterplot
        - **geom_jitter()** a special type of scatterplot, that adds some random noise to points so they don't plot exactly on top of each other
    - **geom_line()** # Line plot
    - **geom_barplot()** # Bar graph

    - **geom_boxplot()** # Boxplots
    - **geom_smooth()** # Trend lines
        - Lots of different kinds of smoothers or trendlines here. The default is loess, which is a wavy curved line
        - The straight line we're all used to is **method = "lm"** for linear model
    - **geom_histogram()** # Histogram
    - **geom_density()** # Smoothed histograms
- You can change aesthetics of specific layers of the plot, by adding 'aes' to the layer you want to customise

Hadley Wickham quote:

-
    - *"In brief, the grammar tells us that a statistical graphic is a mapping from data to aesthetic attributes (colour, shape, size) of geometric objects (points, lines, bars). The plot may also contain statistical transformations of the data and is drawn on a specific coordinates system."*

NOTE 'gg' in ggplot stands for grammar of graphics.

lm = linear model

- So, when we add the **scale_x_log10**, we are altering the 'coordinates system of our plot
- **aes()** function is the "how" – how data is stored, how data is split
- **geom_** is the "what"—what the data looks like. These are geometrical objects stored in subsequent layers.
- can customise your plots by chaging the color pallettes you use, the shapes, the type of lines
    - Colorblind palette: http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/#a-colorblind-friendly-palette
    - Customize shapes and lines: http://www.cookbook-r.com/Graphs/Shapes_and_li2ne_types/
- **theme** is and incredibly powerful way to customise your plot.
    - there are some default themes (e.g., **theme_bw()**), but just using **theme()** will give you way more flexibility
- change axis labels, and plot titles with **xlab()**, **ylab()**, and **ggtitle()**
- **ggsave**(file = "filename") saves the plot
    - can save various formats (png, jpg, pdf, etc)
    - also takes width, height and res arguments
- **geom_jitter** can be used to add random 'scatter' to points in your plot

After Break:

- James's Dropbox R Script: https://www.dropbox.com/s/rnxkuxph4r5od5w/Day2.R?dl=0
- Socrative ROOM NUMBER: UCONNSWC2

Basic programming in R

- Have R start doing stuff on its own
- Has broad application, across progamming languages
- **Condition Statements - have R choose what to do**
- use **if()** to state conditional with { }
- use **else()** with {} to offer alternative to conditional

        - can think of this as a decision tree with 2 options
    - can use other operators for if statements:
        - ">" greater than
        - "<" less than
        - "==" equal to
        - "!=" is not equal to
        - <= less than or equal to
        - >= greater than or equal to
    - can include more than one tesT
        - use **if**(condition){print("statement")} **else if** (condition) {print("statment")} **else**{print("statement")}
        - can combine tests using '&' for 'and' , '|' for 'or'
    - **Loops - let us do repetitive tasks**
    - useful for automating tasks
    - use **for(**increment variable **in** vector**){** perform some task**}**
    - increment variable will get overwritten by each iteration of the for loop
    - **Functions -** helping to do repetitive tasks
    - useful for creating reusable chunks of code
    - You've already been using them in R! For example nrow(dataset) to see number of rows in your data set.
    - Function has an **argument**, which is a variable the function operates on; code within the function; the result of the code output with **return**
        -
            - result <- function(argument)
    - Functions have variable scope: variables named within a function do not alter variables outside of the function, i.e., function variables are 'local' and so do not affect variables in the 'global' environment

#----------#

Day 2 - Afternoon Session

#----------#

Socrative Room ID: FISHER5747

- **Working with dplyr and tidyr and rmarkdown/knitr**
- Lets you manipulate dataframes easily
- We will work with R Markdown notebooks today for which you will need to install these packages:

```
install.packages(c("tidyr", "dplyr", "knitr", "rmarkdown", "formatR"))
```

In R Notebook:
"Knit to HTML" to show all code and data in website

dplyr important functions

1. f**ilter()** to subset rows - generally do this first
2. **select()** to subset columns
3. **group_by()** to count and group data
4. **summarize()** useful information summaries about data
5. **mutate()** add a new column

- This is a pipe: '%>%'. It is a connector between your data and the functions of dplyr (almost like a + in ggplot2)
-
    -
        - One way to think about pipes is "take the data or output on the left side of the pipe and send it to the fuction on the right side of the pipe"
        - The order of operations is dictated by your pipes

group_by() turns a single dataframe into a set of dataframes grouped based on some variable
summarize() can be used to summarize grouped data into a new dataframe with new columns
that are summaries

function **n()** counts the sample size of the variable we grouped by

********** NBNB**********

-
    -
        -
            -
                - Observations = rows
                - Variables = columns

********** NBNB**********

Working with tidyr

- is a different way of reshaping data
- data tables can either be wide or long
    - long format - every row is a single observation of a single variable for a given id
    - wide format - several columns for each id

Gap Wide dataset:https://www.dropbox.com/sh/ae86tlimhtoq48v/AAARAcJ2yGxSE523_q0mV9R3a?dl=0

- Tidy data means that you have columns as variables and columns as observations, where each observation has its own row
- Hadley Wickham's paper explaining the concept of tidy data: http://vita.had.co.nz/papers/tidy-data.pdf
- Use the **gather()** function to convert *wide* data *to long* format
- Use the **separate()** funtion to split variables into separate columns
- Use the **spread()** function to convert from *long to wide*

{R code to convert gap_wide into gap_long)

```
head(gap_wide)

## To make the data long, we will use the function gather()
 library(tidyr)
 gap_long <- gap_wide %>%
 gather(obstype_year, obs_values, -continent, -country)
 dim(gap_long)
```

head(gap_long)

Note: Command+Alt+i inserts a new chunk for you to work with :-)

Setting up an RStudio Project to work with git/github:

- Hadley Wickham (of ggplot2) has a nice help page for git in RStudio
    - http://r-pkgs.had.co.nz/git.html
- If the RStudio project was created with a github repository, skip over the step below
- For a project with no github repository, the first step is to initialize one
    - From terminal
        - Navigate to the project folder using cd and run **git init**
    - From RStudio
        - Click on the project name in the upper right corner
        - Go to Project Options > Git/SVN, and change version control from "none" to "git"
            - Note: if this isn't available, RStudio doesn't know where git is
                - Go to Tools > Global Options > Git/SVN and set where the git executable is
                - You can find out by typing **which git** in a terminal
        - Restart RStudio (follow the prompt to do this).
        - Afterwards, you should have a Git tab in Rstudio
- Once your project has a git repository and RStudio has the Git tab, you can:
    - Stage files by clicking the checkbox beside them in the Git tab
    - Commit stage files by pressing the Commit button and giving a message
    - Diff files to see what you've changed by selecting a file and pressing the Diff button
- To sync your git repository with Github in RStudio
    - Make a new github repository.
    - Use the code github provides to push an existing repository from commandline and run it in a terminal.
    - There should be two lines (yours will vary):
        - **git remote add origin https://github.com/mickley/test2.git**
            - This tells git that there's a remote server it should send things to
        - **git push -u origin master**
            - This line sends your current repository to github
            - The -u option is important, this is what we missed when RStudio didn't work in the workshop
                - It tells the upstream (github) repository to use/sync the same branches we are using
            - After you run this line, the Push and Pull buttons in RStudio's git tab should be enabled
            - For this to work, you need to have already made at least one commit in RStudio
                - (another thing we didn't do in the workshop)

Thank you instructors for a great workshop!! :)