

Lesson 1

Setting up Reproducible Workflow

Start by navigating to a folder that will serve as the host of the main repository, e.g. `/rob/research/projects`
Once there, replicate this folder structure.

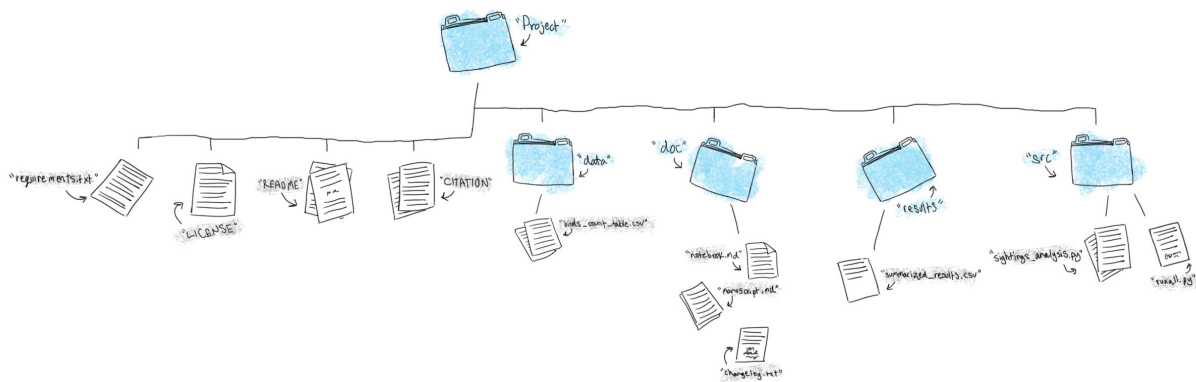


Figure 1:

I recommend doing this on the command line first, just to get some familiarity, but if you want to do it with Windows Explorer or Finder, that's fine too. Here are the folders:

```
mkdir myrepo
cd myrepo
mkdir doc
mkdir src
mkdir results
mkdir data
```

Now make some files at the root level of the directory:

```
touch LICENSE
touch README
touch CITATION
touch requirements.txt
```

Ok - all done - for now. Let's talk about what goes into each, and then we'll start to populate the folders a bit.

Start With Raw Data

I've given you a raw dataset from the Bahamas Marine Mammal Research Organization. They are in csv format. Download them from box, and put them into your `data` folder. You can look at them with the `head` command. Note where you are on the command line first, otherwise you may be looking for data in all the wrong places.

```
head bbmroData.csv --lines=2
```

A Few Scripts

Ok, at this point we have a file structure, and raw data - nothing else. But let's keep going with a simple R script to read in the data. Make a script in the `src` folder and call it something clever like `readData.R`. Try this first on the command line.

```
touch readData.R
```

While we're at it, let's make three more empty files:

```
touch summarizeData.R  
touch plotData.R  
touch runRegression.R
```

And then we'll make a controller script, since that will be useful.

```
touch runAll.R
```

That will make the files, but they're going to be empty just now.

Lesson 2

Awesome. Now we're ready to move things along with git somewhat. At this point, we've made a structure that's consistent, and can form the basics of a reproducible workflow. We've talked about one way to manually track the changes of the whole project - namely taking the whole structure, and putting it into a folder with a date-specific name, e.g. `20170-08-06-contents-of-myRepo`

This can work, and is not a bad setup, but it requires a lot of intervention and consistency on the part of you the project organizer. One other drawback - in my mind - is that with more than one user and/or more than one computer, it's very hard to scale. Things can get complicated and out of phase/sync quickly. The solution? A modern version control system; here we'll explore `git`

`git` was written by Linus Torvalds - the creator of Linux. It was an outflow of bitkeeper, and was built to manage the linux kernel. Undoubtedly this is a bigger project than you or I will ever work on. At present the kernel is approximately 17 million lines of code, with 2-3 thousand contributing developers. No wonder they need a robust system. We've talked about what `git` is in the lecture, and now we'll start to put it into practice.

Let's `cd` into the `src` directory. Type `pwd` just to make sure you are where you think you are. Then let's initialize the repository:

```
git init
```

If successful, all you'll get a message indicating that you have initialized an empty repository. Success!

Next let's look at the status:

```
git status
```

You will see a message about being on branch master, that you are on the initial commit, and that there is nothing to commit. Ok, so what now? Let's add a file to the staging area. Recall that there are three states of git:

1. the working directory
2. the staging area
3. and the git repo itself - i.e. where things end up when you commit

Right now we are in the staging area, and we should have 5 `*.R` files - 4 that will do something eventually, and one that will run all the others. Let's add the first one `readData.R` (or whatever you named the file to read in the data).

```
git add readData.R
```

You should get nothing back - just the command prompt again. But if you ask for the status again, you'll see something different:

```
git status
```

Now that we've added a file to the staging area, we should see a message indicating there are changes to be committed, and it should label what the new file is. On my computer it looks like this:

Now that we've added it, let's run our first commit:

```
git commit -m "First Check-in of readData.R script"
```

Now we get some more information back from git about this commit, including the all-important sha-1 identifier. Here you only see the abbreviated sha-1, but if you type `git log` you'll see the full sha-1:

```
git log
```

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/gitTemp/src$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   readData.R

rob@rob-Precision-5510:~/Documents/business/2017_ESA/gitTemp/src$
```

Figure 1:

Along with who made the commit (hopefully you), the date, and the commit message. Now let's do some comparison between git and the file system. Right now we have several R scripts in the directory. What do you think you'll get when you type these commands?

Think what the answer will be before you issue the command.

```
ls
git ls-files
```

Why the difference? What do you think you'll see if you ask for the status? Think before you issue the command:

```
git status
```

If you have untracked files, why is it that git knows about them?

Going through the Cycle Again

Let's edit the file now to actually read in the data. You can do this in vi, or in the text editor of your choice.

Here's what I added to mine (recall the tip to put at least a barebone comment at the start of the script):

```
## This script will read in raw data from the Bahamas Marine Mammal Research Organization
## into a data frame called whales. whales will serve as the intermediate data for
## subsequent analysis
whales <- read.csv(file = '../data/bbmroData.csv')
head(whales)
```

Now we have some new code, i.e. the file has been modified. So go back to the command prompt, and type:

```
git status
```

What do you see, and why is it different from whatn you issued git status a few lines back? What do you do next if you want to make a snapshot (commit) of the updated file? Add it and commit it, but this time we can do what is called an express commit. Because git already knows about `readData.R` we can combine the add and the commit steps:

```
git commit -am "Add Code to Ingest the Raw Data"
```

But, this can only be done for files that have already been added.

Let's go ahead and add all of the remaining untracked files and commit them:

```
git add .
git commit -m "Add Remaining Blank R Files"
git status
```

Your last command should indicate that you have a clean repo. Cool!

Git Log

Now that we've made a few commits, we can start to look at how our project is evolving. Git log is a good way to do this; we've seen this before, but now we'll see a bit more

```
git log
```

This is a very flexible command, however, and there are a few ways we can clean it up to make the output a bit tidier:

```
git log --oneline --decorate --graph
```

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git log --oneline --decorate --graph
* 0a215c7 (HEAD, master) Add Remaining R Files
* 87f7030 Add New Line
* 48f84c5 Update Code to Read in Whale Data
* 9efbc59 First Check-in of readData.R Script
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$
```

So what all have we asked for? We've said, show us the log, but in a compact form, decorate it to have it show where the current branch (master) is pointing (HEAD), and show us the graph. Recall that git is a directed graph, where snapshots (commits) are the nodes. On the graph above we have 4 nodes, with the master branch pointing to the latest commit. We can move this pointer, but that's a topic we'll take up later.

If we want to really get a good sense of what things look like at the different nodes, use the `git show` command.

```
git show
```

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git show
commit 0a215c7adb4c49ec863d4d02dcb0a76998d63f04
Author: robschick <robschick@gmail.com>
Date: Fri Aug 4 23:04:18 2017 -0400

    Add Remaining R Files

diff --git a/plotData.R b/plotData.R
new file mode 100644
index 0000000..e69de29
diff --git a/runAll.R b/runAll.R
new file mode 100644
index 0000000..e69de29
diff --git a/runRegression.R b/runRegression.R
new file mode 100644
index 0000000..e69de29
diff --git a/summarizeData.R b/summarizeData.R
new file mode 100644
index 0000000..e69de29
```

And then one specific node:

```
git show 87f7030
```

Once we've asked for a specific node, we get a lot of useful information about the commit, including who did it, when they did it, what the message was, and what the commit contained.

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git show 87f7030
commit 87f7030822cefc039fbf59796fd5d475f3fc3980
Author: robschick <robschick@gmail.com>
Date:   Fri Aug 4 23:01:07 2017 -0400

    Add New Line

diff --git a/readData.R b/readData.R
index b363e62..99fe003 100644
--- a/readData.R
+++ b/readData.R
@@ -3,3 +3,4 @@
 #' frame will serve as the intermediate data for subsequent analysis
 whales <- read.csv(file = '../data/bbmroData.csv')
 head(whales)
+
```

Figure 2:

Commit Messages

Recall what we talked about in Lecture - good commit messages are worth the effort, and will help you come back to the repo/project and understand not only what you did (use the `diffs` for that), but also why. If you are writing command line commit messages, it's hard to really write a thoughtful one. However, if you have an editor configured (my default is `vi`) then you can easily write a good one. Try it (with the knowledge that this commit is really a toy commit). You can invoke the editor by just typing `git commit` without any flag. Here's what I get when I type that:

```
COMMIT_EDITMSG (~/Documents/business/2017_ESA/demorepo/src/.git) - VIM
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   modified:   plotData.R
#
~
~
~
```

Figure 3:

Now you can type a good message with a less than 50 character Capitalized First Line, followed by detailed text that describes the why of the commit. With a declarative succinct first line, you can see at a glance what each commit is about, and then with `git show` you can dig in and read more.

On the subject of declarative, here's a useful tip. You want the first line to complete this sentence = 'If added, this commit will ...'

Here's what I get - note the top part of the figure that shows the oneline with the declarative statement. . . "If added, this commit will Add a Draft Histogram of the Sea Surface Temperature Data."

```

rob@rob-Precision-5510: ~/Documents/business/2017_ESA/demorepo/src
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git log --oneline --graph --decorate
* 08ea22d (HEAD, master) Add A Draft Histogram of the Sea Surface Temperature Data
* 0a215c7 Add Remaining R Files
* 87f7030 Add New Line
* 48f84c5 Update Code to Read in Whale Data
* 9efbc59 First Check-in of readData.R Script
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git show 08ea22d
commit 08ea22d286ab1fe434309418b5e972443d7472dc
Author: robschick <robschick@gmail.com>
Date: Fri Aug 4 23:16:53 2017 -0400

    Add A Draft Histogram of the Sea Surface Temperature Data

    I'm operation under the assumption that we want to see how the
    temperatures in which the animals are distributed will be a good first
    attempt at EDA for this project. Accordingly, I'm starting with a
    histogram of the SST data.

diff --git a/plotData.R b/plotData.R
index e69de29..b1b74dd 100644
--- a/plotData.R
+++ b/plotData.R
@@ -0,0 +1,2 @@
+#' Plot out the data
+hist(whales$SST)
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$

```

Figure 4:

Git Diff

How can we easily see what changed between two commits? We may see that one line indicated a big change that we want to inspect. Use the diff command on any two nodes in the graph:

```
git diff 9efbc59 48f84c5
```

Note that the order matters. The code above says show me what changed as the repo moved from 9efbc59 to 48f84c5 Here's what mine looked like:

```

rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git diff 9efbc59 48f84c5
diff --git a/readData.R b/readData.R
index e69de29..b363e62 100644
--- a/readData.R
+++ b/readData.R
@@ -0,0 +1,5 @@
+#' This script will read in the raw data from the Bahamas Marine Mammal
+#' Research Organization into a data frame called 'whales'. This data
+#' frame will serve as the intermediate data for subsequent analysis
+whales <- read.csv(file = '../data/bmroData.csv')
+head(whales)
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$

```

Figure 5:

File Management with Git - rm and mv

One thing that is good to remember is that you've added a file to git, then it's being tracked until you delete it. And actually even if you delete it, it gets stored in previous commits - because it *was* in that repo at one point in time. Though the file is just a file, once it's tracked, don't be tempted to remove it via the operating system.

If you want to delete a file, or if you want to change its name, git offers utilities to do this. Not surprisingly, they are `git rm` and `git mv`. Let's see them in operation. First we'll add two files - one to move/rename and one to delete. We'll do this in the `src` directory again.

```
touch prediction.R
touch simData.R
git add .
git commit -m "Add Two Files for Experimenting with Moving and Deleting"
```

With these setup and added, let's delete it at the command line just to see what happens:

```
rm prediction.R
```

All looks good right from the OS point of view - right? What happens when you ask `git` what the status is?

```
git status
```

What to do now? We back out the change

```
git checkout -- prediction.R
git status
```

Now we're back to a clean working directory. So we can delete with `git`. Not that when you do this, and look at the status, it looks very different - the first one makes a change (the deletion) that hasn't been staged, where as the second (`git rm`) makes a change that has been committed and is ready to stage:

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ rm prediction.R
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    prediction.R

no changes added to commit (use "git add" and/or "git commit -a")
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git checkout -- prediction.R
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git status
On branch master
nothing to commit, working directory clean
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git rm prediction.R
rm 'prediction.R'
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    prediction.R

rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$
```

Basically to get a clean working directory if you go the `rm` route, you have to add the file to the staging area, only to then commit it. `git rm` just does that for you in one step.

Now is a good time to commit the deletion.

```
git commit -m "Delete prediction.R File"
```

Now that file is gone and the repo is clean, let's rename the other file - this time using `git`. As above, if you use `git`'s utilities for this, you don't have to add the changes

```
git mv simData.R simXYData.R
git commit -m "Rename Simulation Script for Clarity"
```


The .gitignore File

One person in the responses to the survey asked about what to version control and what not to. I tend to keep just the code versioned, and things that can be easily created on the fly, I ignore. If you create a lot of these files, e.g. intermediate *.Rdata files, or *.png files, etc., your repo can get pretty messy as you create them. Even if you aren't tracking them, git is aware of them, so typing git status can yield a lot of information.

Thankfully, there's a mechanism, and it's called the .gitignore file. All you have to do is create it, put in a few patterns to match (and thus have git ignore), and then place it into version control. Let's try it with a pdf file. First add the plotting device code to your plotData.R script.

```
pdf(file = '../results/firstHistogram.pdf')
hist(whales$SST)
dev.off()
```

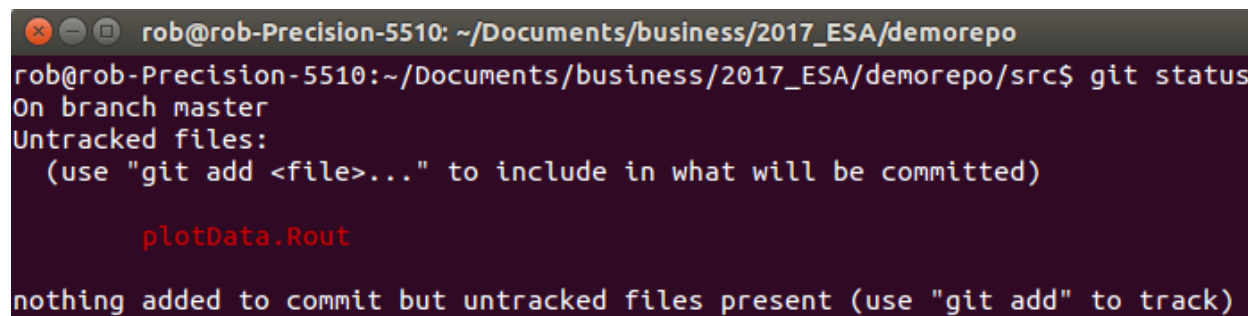
Add and commit these changes, and then let's run the code from the command line.

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ vi plotData.R
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git commit -am "Add Plotting Device to Create a PDF"
[master b3dced] Add Plotting Device to Create a PDF
1 file changed, 2 insertions(+)
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ R CMD BATCH --vanilla plotData.R
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$
```

What do you think we'll have now? How many new files? Where and what will they be?

If you answered 2 files - gold star for you. There's the .pdf file in the results directory, but there's also the .Rout file in the src directory. But if you type git status what do you think you'll see?

Probably not what you expected to see, at least it wasn't what I thought we'd see.

A terminal window screenshot showing the output of the 'git status' command. The window title is 'rob@rob-Precision-5510: ~/Documents/business/2017_ESA/demorepo'. The terminal text shows: 'rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src\$ git status', 'On branch master', 'Untracked files:', '(use "git add <file>..." to include in what will be committed)', 'plotData.Rout', and 'nothing added to commit but untracked files present (use "git add" to track)'.

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        plotData.Rout

nothing added to commit but untracked files present (use "git add" to track)
```

Figure 6:

But here's a good thing to be aware of. We only have the repo initialized in src, and not in results. At any rate, there's a chance we might create the odd pdf file here in addition to the .Rout files, so let's exclude them both. We do this with the .gitignore file, which we have to make, and which, we have to place under version control.

To do this, make a new text file, and add these two lines:

```
*.Rout
*.pdf
```

Save it as .gitignore, then add it, and then commit it.

Now let's rerun the R script that will make the .Rout file and we'll see what happens. Before you type git status what do you think you'll see?

If all goes well you'll not see anything - git will indicate that you have a clean working directory.

Time for a Break

Before we move on to replicating this in RStudio, it's probably time for a break.



Figure 7:

Lesson 3 - Git in RStudio & Time Travel

The learning objectives here are 1) to work through the code/add/commit cycle in RStudio, and 2) to get a feel for how you can move around the repository. This is pretty mind-bendy to me at times, but if you go back to the core idea of git storing a series of snapshots as nodes along a graph, you'll be able to wrap your head around the idea of revisiting any of those nodes at any time.

RStudio

RStudio provides a very nice front-end IDE for R. As it continues to develop, more and more capabilities are added. Version control with `svn` or `git` has been available for quite some time. You may have even seen the git tab in RStudio and wondered what it was doing there. I use both the command line and RStudio for my git work, so it's a good idea to have a feel for both. I prefer the command line interface for some things, but being a visual person, RStudio's git support has a lot to offer. IN particular, it's quite easy to write a good commit message, and to visually see the diffs/changes that you are making.

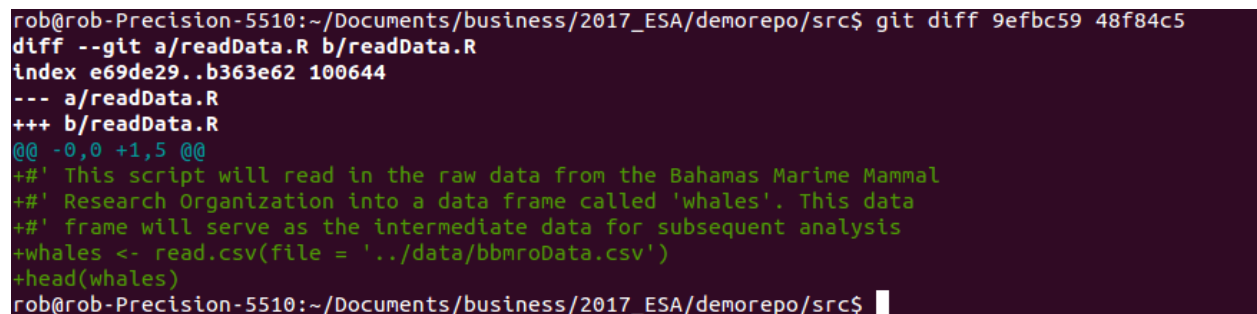
You will have seen how we did this in the Lecture - now it's your turn. Make some changes to a file, add and commit them. Also, add a new script, and note the difference in the buttons in the git tab in RStudio - i.e. when do you see an A in the button vs. an M? What are the command line analogs?

Git Diff

How can we easily see what changed between two commits? We may see that one line indicated a big change that we want to inspect. Use the diff command on any two nodes in the graph:

```
git diff 9efbc59 48f84c5
```

Note that the order matters. The code above says show me what changed as the repo moved from 9efbc59 to 48f84c5 Here's what mine looked like:



```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git diff 9efbc59 48f84c5
diff --git a/readData.R b/readData.R
index e69de29..b363e62 100644
--- a/readData.R
+++ b/readData.R
@@ -0,0 +1,5 @@
+#' This script will read in the raw data from the Bahamas Marine Mammal
+#' Research Organization into a data frame called 'whales'. This data
+#' frame will serve as the intermediate data for subsequent analysis
+whales <- read.csv(file = '../data/bbmroData.csv')
+head(whales)
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$
```

Figure 1:

When you think about working with your future self - let alone any additional collaborators - these diffs are powerful ways to see what actually changed. Couple the diffs with a good commit message, and you can quickly get oriented into how and why things change over time.

Compare this to a script that you just rename, or keep saving as you alter it. You may get to a point with that script, where it's broken and it's hard to go back to its state when it last worked. Or you may recall having ventured down a pathway that you gave up on, only *now* you want to get back to that path to try that experiment again. With `git` this is straightforward. Without `git` it's just about impossible.

One thing you can get in the habit of doing (if it works for you) is to look at the diffs before you commit. My typical workflow is something like:

1. write some code
2. Add it
3. Commit it
4. write some more code, etc.

Before you add and commit the changes, though, you can just type `git diff` to see what has been changed. With the code we've been running here, it's just toy syntax, but if you were working on a complicated function, it's often nice to see what you've done.

Going to a Particular Snapshot

Now that we have a few commits built up, we can go back to a particular commit and have our code (in the same file!) look exactly as it did at that commit. And we can do this without throwing away the most recent work. Take a look at your history with `git log --oneline` and chose a commit you want to navigate to.

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git log --oneline --decorate --graph
* e028cd2 (HEAD, master) Check-in .gitignore File
* b3dccc2 Add Plotting Device to Create a PDF
* c793018 Rename Script for Clarity
* c5db5d6 Delete prediction.R file
* 66eadc0 Add Two Files for Experimenting with Moving and Deleting
* 08ea22d Add A Draft Histogram of the Sea Surface Temperature Data
* 0a215c7 Add Remaining R Files
* 87f7030 Add New Line
* 48f84c5 Update Code to Read in Whale Data
* 9efbc59 First Check-in of readData.R Script
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$
```

Figure 2:

Yours will be different from mine, but to go to that node, we use the `checkout` command:

```
git checkout 0a215c7
```

You'll see the detached HEAD warning, and if you type `git ls-files` you'll see the files git was tracking at that point in time. If you look at the plot script, you'll see it at that state:

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git checkout 0a215c7
Note: checking out '0a215c7'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 0a215c7... Add Remaining R Files
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ more plotData.R
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$
```

Figure 3:

In my case the plot script.

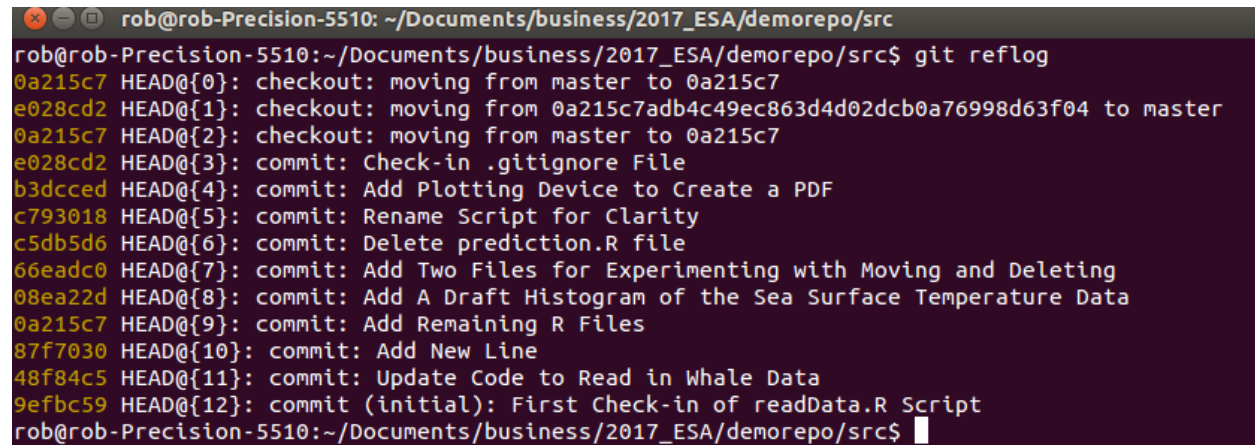
Two thoughts here:

1. If we wanted to recreate old figures. This is one way we could do it. Checkout an old repo, and re-issue the R code from the command line to make the file. You can also hang on to them as we mentioned in

lecture via the storing of old copies of the `results/Figures` folder with an archival date. Just make sure you are consistent if you follow this, otherwise you may end up with an old file that has a current name, but not the current content, etc.

2. Type `git log --oneline --decorate` here to see what you get. You'll probably see a shortened history, because we've moved the pointer back to this particular commit, and commits know their parent commit - so the later ones aren't listed. This may give you great pause because you don't see work you've already done. There's yet another command to help you.

```
git reflog
```

A terminal window showing the output of the `git reflog` command. The terminal title is `rob@rob-Precision-5510: ~/Documents/business/2017_ESA/demorepo/src`. The output lists 13 HEAD positions with their corresponding commit hashes and descriptions:

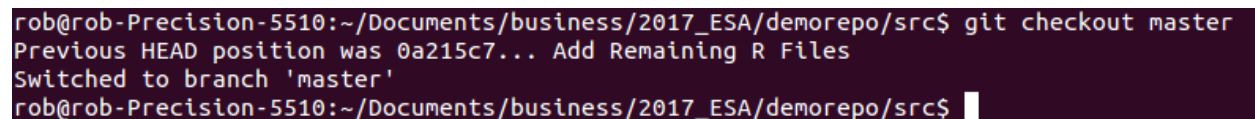
```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git reflog
0a215c7 HEAD@{0}: checkout: moving from master to 0a215c7
e028cd2 HEAD@{1}: checkout: moving from 0a215c7adb4c49ec863d4d02dcb0a76998d63f04 to master
0a215c7 HEAD@{2}: checkout: moving from master to 0a215c7
e028cd2 HEAD@{3}: commit: Check-in .gitignore File
b3dccc HEAD@{4}: commit: Add Plotting Device to Create a PDF
c793018 HEAD@{5}: commit: Rename Script for Clarity
c5db5d6 HEAD@{6}: commit: Delete prediction.R file
66eadc0 HEAD@{7}: commit: Add Two Files for Experimenting with Moving and Deleting
08ea22d HEAD@{8}: commit: Add A Draft Histogram of the Sea Surface Temperature Data
0a215c7 HEAD@{9}: commit: Add Remaining R Files
87f7030 HEAD@{10}: commit: Add New Line
48f84c5 HEAD@{11}: commit: Update Code to Read in Whale Data
9efbc59 HEAD@{12}: commit (initial): First Check-in of readData.R Script
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$
```

This command basically stores every command you issue in git. So you can see that now we're checked out on the `0a215c7` node, and even though we don't see all the other downstream commits when we type `git log` we can rest assured that they commits are all still there.

This can be really useful as you progress and get into more complicated things with merging branches and rebasing the repository. The take home is that git tries really really hard not to lose your data.

To get back to the last commit:

```
git checkout master
```

A terminal window showing the output of the `git checkout master` command. The terminal title is `rob@rob-Precision-5510: ~/Documents/business/2017_ESA/demorepo/src`. The output shows the previous HEAD position and the branch being switched to:

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git checkout master
Previous HEAD position was 0a215c7... Add Remaining R Files
Switched to branch 'master'
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$
```

Figure 4:

Lesson 4 - Working Remotely

We're almost done! Phew. We've covered a lot of ground, but there's still another important lesson, and that is embracing what some people call the 4th state of git - the remote repository. There are many ways to do this, and sometimes they all seem sort of similar but different in frustrating ways, i.e. you might think you are following a work-flow that worked for you only to run into some weird unexplained git error.

Important note here - git is huge and can be frustrating - but since it's huge and since it was developed by an unber-nerdy crowd, there are a huge amount of resources out there. My first thing to do if I see an error I don't understand, is to type it into google, and start browsing the answers on StackOverflow. At current count, there are 88,555 questions tagged with `git` on stack overflow. Chances are, the answer is out there!

At any rate, one workflow to connect a local repo to GitHub is as follows (n.b. this is done at the start):

1. on github, add a new repository. I typically add a R themed `.gitignore` file, and a `README`
2. copy its url (`http` or `ssh` - I prefer `ssh`) to clone it
3. Navigate to the parent folder, clone the repo
4. At this point you can either `cd` into the repo and start work, or if you want to work in RStudio, simply start a new project with the "Existing Directory" option chose the repo and the git tab will appear.

You can also bypass the cloning from the command line be choosing the option to clone from a repo, but make sure all your `ssh` keys/ducks are in a row.

Now when you look at the git tab in RStudio, you will see one important difference from before - the Push/Pull tabs are live, i.e. they are not greyed out.

However, what if you already have a local repo that is not connected to GitHub? Sort of like we have here. First type this:

```
git remote -v
```

You should see nothing! How do we connect to github then? Follow these steps:

1. Go to github and make a new empty repo using the same name as the one on your computer (you can use different names, but let's keep it simple for now).
2. Copy the url of the repo - using the `ssh` option
3. Add the remote locally
4. Verify the connections
5. Push the changes.

Let's see that:

Once you click on Create Repository, GitHub gives you this *really* helpful page:

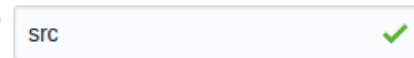
Create a new repository

A repository contains all the files for your project, including the revision history.

Owner





Repository name



Great repository names are short and memorable. Need inspiration? How about **ideal-umbrella**.

Description (optional)


 **Public**
Anyone can see this repository. You choose who can commit.

 **Private**
You choose who can see and commit to this repository.

Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾

Add a license: **None** ▾ 

Create repository

Figure 1:

robschick / src

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Settings Insights

Quick setup — if you've done this kind of thing before

or **HTTPS** **SSH** `https://github.com/robschick/src.git`

We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# src" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/robschick/src.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/robschick/src.git
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

In our case, we're going to push the existing repo from the command line (note that of course your url will be different):

```
git remote add origin https://github.com/robschick/src.git
```

Once we've added it, let's look again to see if we are talking to github:

```
git remote -v
```

I see this now:

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git remote -v
origin https://github.com/robshick/src.git (fetch)
origin https://github.com/robshick/src.git (push)
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$
```

Figure 2:

With it added, then let's push our changes

```
git push -u origin master
```

Wait, why did it fail?

I know I used the right password, but it's failing. Well, you'll note in my enumerated list that I said I prefer using SSH - this is why. If I look at my config file I'll see that in the remote.origin.url it's using https, but I want ssh:

I change it to ssh at the command line:


```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git push origin master
Username for 'https://github.com': robschick
Password for 'https://robschick@github.com':
remote: Invalid username or password.
fatal: Authentication failed for 'https://github.com/robshick/src.git/'
```

Figure 3:

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git config --list
user.name=robschick
user.email=robschick@gmail.com
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=https://github.com/robshick/src.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$
```

Figure 4:

```
git remote set-url origin git@github.com:robschick/src.git
```

You can verify that it is right with

```
git remote -v
```

And then happiness ensues:

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git push origin master
Warning: Permanently added the RSA host key for IP address '192.30.255.112' to the list of known hosts.
Counting objects: 26, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (20/20), done.
Writing objects: 100% (26/26), 2.66 KiB | 0 bytes/s, done.
Total 26 (delta 7), reused 0 (delta 0)
remote: Resolving deltas: 100% (7/7), done.
To git@github.com:robschick/src.git
 * [new branch]      master -> master
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$
```

Figure 5:

And finally we can see them on GitHub to make sure all is right.

Pushing Local Changes

So now we've pushed up all of the changes we have made to date locally, so they are mirrored on GitHub. One of the biggest advantages of using git, or any distributed version control system, is that you now have multiple points of failure, instead of just one, i.e. you've increased your bus factor and this is always a good thing.

When we work with a remote repo, we now add an extra stage to our local git workflow. From time to time, we need to push our local changes to the local repository. How often you do this is a matter of personal preference. I tend to do it a lot, especially since my code is typically only being used by me. For someone like Hadley Wickham who is writing a lot of influential R code, his feeling is that pushing is essentially publishing. So each time you push, it'd better work!

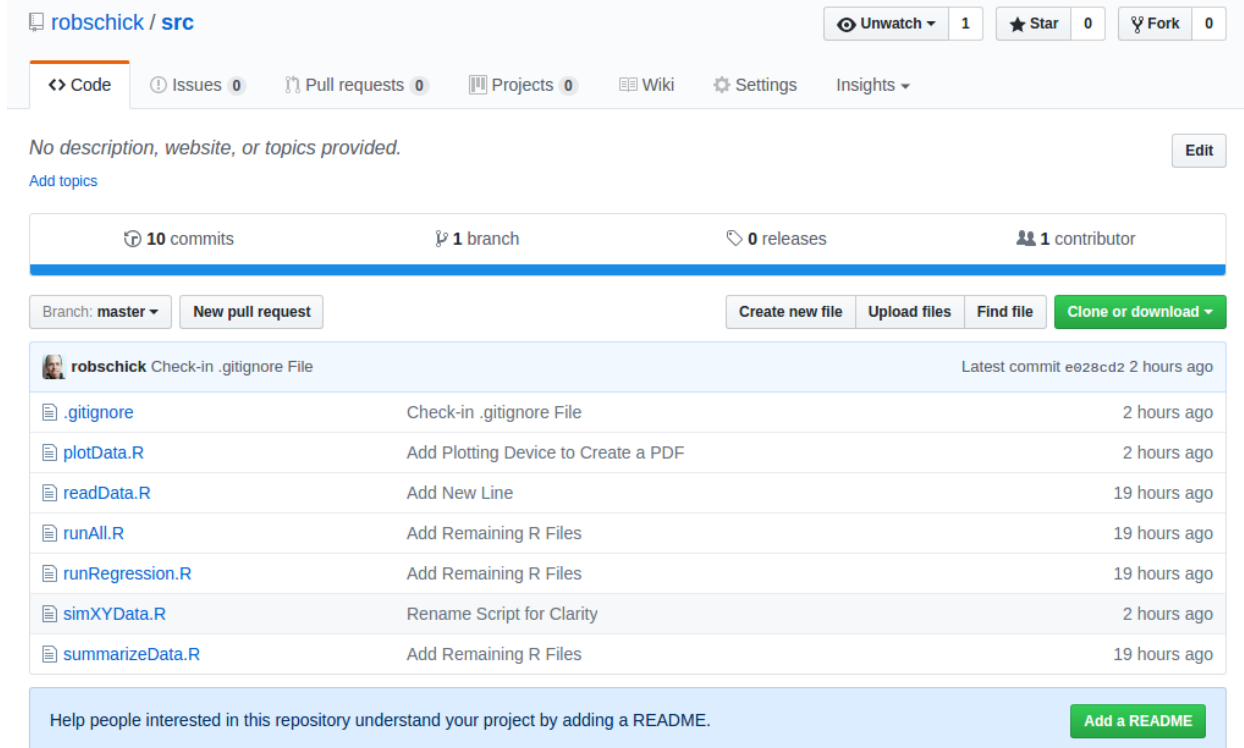


Figure 6:

Go ahead and repeat the cycle - either in RStudio, or the command line. Add some code, add some files, commit them, and push them. Then go and view them on GitHub.

Pulling Changes

One best practice is issue `git pull` before you push. This just ensures that you have the most up to date code before you push. The benefits of this will become apparent in the next section

Merge Conflict

Ah the dreaded merge conflict. These can be a pain in the butt when you first encounter them, but all in all, they are pretty straightforward. As you get more advanced with git, you can configure a graphical merge tool if you want to help with this. They can be great, but for now, we'll just handle this on the command line.

What is a merge conflict? It's essentially what happens when the repository gets out of sync. Let's say you make a change on your local computer, and push it to github, but then you working on a node, and you make a change there - but you haven't yet brought down the most up to date changes from GitHub. When you go to push your changes up to the node, you will get rejected, and then you have to manually resolve the conflict.

We don't have a node here, but we can mimic it well enough with a local repo and GitHub:

1. In your repo, make a change to a file, add it, and commit it, but don't push it to GitHub.
2. Then, navigate to github, and make a change to the same file. Use the pencil icon to edit it directly
3. Add a commit message, and commit the change

4. Navigate back to the command line and issue the `git push` command
5. If you are lucky, all hell will break loose

Here's what I have on GitHub

The screenshot shows a GitHub commit page for the file `src / readData.R`. The commit was made by `robschick` with the message "Add Species Name Tally to readData.R" on the `master` branch. The commit hash is `57a18ab` and it was made "just now". There is one contributor. The file details show it has 7 lines (6 sloc) and 301 Bytes. The commit message is "Add Code To Tally the Counts". The diff shows the following changes:

```
diff --git a/readData.R b/readData.R
index 99fe003..19dd11d 100644
--- a/readData.R
+++ b/readData.R
@@ -3,4 +3,5 @@
 #' frame will serve as the intermediate data for subsequent analysis
 whales <- read.csv(file = '../data/bbmroData.csv')
 head(whales)
+table(whales$count)
```

Figure 7:

And what I have locally:

```
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git show
commit 6bfe74cac0d86f9c2dc6c38c444a8825615972b8
Author: robschick <robschick@gmail.com>
Date: Sat Aug 5 18:17:04 2017 -0400

    Add Code To Tally the Counts

diff --git a/readData.R b/readData.R
index 99fe003..19dd11d 100644
--- a/readData.R
+++ b/readData.R
@@ -3,4 +3,5 @@
 #' frame will serve as the intermediate data for subsequent analysis
 whales <- read.csv(file = '../data/bbmroData.csv')
 head(whales)
+table(whales$count)
```

Figure 8:

Ok, let's give it a whirl:

```
git push origin master
```

Sad times, but we can fix the problem. First, let's pull.

```
git pull origin master
```

And here we see the conflict:

```

rob@rob-Precision-5510: ~/Documents/business/2017_ESA/demorepo/src
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git push origin master
To git@github.com:robschick/src.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@github.com:robschick/src.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ █

```

Figure 9:

```

rob@rob-Precision-5510: ~/Documents/business/2017_ESA/demorepo/src
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git pull origin master
From github.com:robschick/src
 * branch                master      -> FETCH_HEAD
Auto-merging readData.R
CONFLICT (content): Merge conflict in readData.R
Automatic merge failed; fix conflicts and then commit the result.
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ █

```

Now we have to fix them manually, which we do in a text editor

```

readData.R (~/Documents/business/2017_ESA/demorepo/src) - VIM
''' This script will read in the raw data from the Bahamas Marime Mammal
#' Research Organization into a data frame called 'whales'. This data
#' frame will serve as the intermediate data for subsequent analysis
whales <- read.csv(file = '../data/bbmroData.csv')
head(whales)
<<<<<<< HEAD
table(whales$count)

=====
table(whales$speciesName)
>>>>>>> 57a18ab1334773b2033b00edc2f4de771346366c
~
~

```

Figure 10:

Once it's resolved, then we go through the regular cycle again:

```

rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ vi readData.R
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git add readData.R
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git commit -m "Resolve Merge Conflict"
[master 2cf2829] Resolve Merge Conflict with GitHub Code
rob@rob-Precision-5510:~/Documents/business/2017_ESA/demorepo/src$ git push origin master

```