Intro to Git

We've talked at a very basic level about how to work with the computer, but today we're going to start talking about how to put together actual projects - complex works with many moving pieces and many collaborators. Git is one basic tool that helps facilitate large projects.

We are all used to working in Microsoft word, saving as we go. But it's relatively difficult in Word to access older versions of your document. Say, after weeks of work you decide you want to get rid of several days worth of edits. Very difficult to do - they're all embedded in the one document. Git is a piece of software that enables you to get past this limitation. It enables what we call version controlling, it's a way of controlling and maintaining different versions of your software at one time.

When we use git we tell the computer to take snapshots of our work at different times throughout it's history. We work with files until we decide we want to take a snapshot of them. Then we take a snapshot and commit it to the memory of the database. At any time we can go back and recover virtually any stage of the process. It makes working much less scary. Nothing is lost forever, because we can easily recover files from different points in the past. Because we have full versions of the code at various points in time we can also do some fairly fancy tricks where we allow different versions of the code to exist alongside each other.

Also allows people to collaborate easily.

There is a particular workflow that we characterizes this use of version controlling.

Workflow:  
Modify a file (in a text editor)

Add that file to the staging area ($git add file\_name)

Commit it to the repository ($git commit)

Work some more.

The reason to have a workflow like this is because you don't want to save EVERYTHING. Sometimes you want to just screw around to get something to work. But when you're satisfied that you're ready, you have to get things through to the commit stage before they're actually saved permanently to the history of the repository. Until then, they're in git's short-term memory.

Studied example.

Yesterday we worked with HTML. Today we are going to wave at the work we did yesterday. We'll be using HTML/CSS again, but we won't think too much about what we're typing. We're going to focus on the Git by copy and pasting the HTML. So let's make a folder my\_website using the terminal and change into it.

$ mkdir my\_website

$ cd my\_website

From inside the directory you just made:

A folder is a folder until you make it git-ready. Then it becomes a repository. We have to tell the computer that we want to use git to trace the version history of a folder. So the first thing to do is "initialize" a git repository. Intuitive CLI FTW:

$ git init

You should get some feedback saying you initialized an empty repository. Success! We have a folder with nothing in it. Like gods, the world is a blank canvas for us to play with. So now we have set up our staging area. Create a file index.html

$ touch index.html

Now copy and paste this text into it.

<https://raw.githubusercontent.com/bmw9t/git_intro_hilt/d2c40e75cbe30e937e9c2253a8e80e21dca64ecb/index.html>

This should look familiar. It's our basic HTML framework. What else is going here? Right. I've stolen a page by Eric Rochester, one of the gurus at the scholars' lab. Theft is the highest form of flattery, after all. So paste it in, save it, and take a look at this new command:

$ git status

Update the working tree. We've modified something but not staged it.

Pretty intuitive - it tells us what is going on in our git project. And it tells us what to do next - add the files so that they will be tracked and staged. Let's do that and hit status again.

$ git add index.html  
$ git status

Now we have things staged but not committed just yet.   
  
$ git commit -m "My initial website framework."

The git commit command, obviously, commits our changes to the snapshot history of the repository. It stores it in the history of the repo. The -m let's us give it a message. We need to give detailed messages so that we can tell what changes we made at which points in the history of the repository. Simply having different chunks of code is useful, but not as useful as giving human readable notes about the changes you've made.

Success! Now we have finished the process of committing, and the working area goes back to the beginning. We also get a dot on the git tree.

Let's keep going. This page is terrible so let's give it some style.

https://raw.githubusercontent.com/bmw9t/git\_intro\_hilt/eaa721eab4aa01d7526c47d4da258f2556d0d1b6/index.html

Where are we in the workflow? We've modified. What happens next? We stage. How do we do that?   
  
$ git add .

The . tells the terminal to apply the command to everything in the directory. So it's a shorthand for saying "add everything." Now we stage.

$ git commit -m "Added some style."

Modify the tree.

Next edit.

<https://raw.githubusercontent.com/bmw9t/git_intro_hilt/4a9bea32b32b3008039fe4b31e472de486017ca4/bio_corrected.html>

Before we move on, let's learn a new command. The changes here are more subtle, so let's say that we want to know what they are.

$ git diff

tells us what we have modified. It works by lines added and lines removed. Hence the positive and negative signs on the left margin. So looking down that column we can see what has changed.

Let's go through and commit this - add, commit. At this point we have three commits, three distinct moments in the history of the project. Things are getting complicated. Say that we want to know what has changed up to this point.

$ git log

tells us the history of our commits to this point. Three commits with our beautiful messages. We can also reproduce the beautiful tree that we've been building (important in a moment).

$ git log --graph

One thing to note here is to note that the log tells us what we've done in reverse order.

One last thing before we blow your minds.

https://raw.githubusercontent.com/bmw9t/git\_intro\_hilt/8879c3cb7b81da491ba9bd8eccf253d43f532578/index.html

You tell me what to do. What do I modify on here.

The -m option allows us to leave a message when we commit.

$ git commit -m "Added lorem ipsum to the website."

Lorem ipsum is standard junk text used by typesetters throughout the centuries as they prepare their work.

~18

**Branches**

Because Git takes snapshots of all our code at different times, it is very easy to allow different versions of the same code to co-exist. Say we think we might want to try something different, but we're not certain that we want to go with the new option. Or, say we are working on a project with many collaborators. Git makes it easy for you to break up the project and have people working on very small tasks. You work on the individual thing until it's ready for primetime. This is where things get (even more) confusing. To go with our tree metaphor, our next command is

$ git branch indent

So what this does is take the central tree and split it (gives it a branch) that we tell the computer to name indent. The branch name usually identifies what we want the new feature to be. In this case, we're going to try out some new formatting. Important to note here is that this new branch exists at the same time as the main tree (called master). It's really a lot like time travel. We're creating a parallel universe in which our text is indented. Let's move into that parallel universe

$ git checkout indent

Now we are no longer in the main world. We are in the land of indentations.

So, the link:

https://raw.githubusercontent.com/bmw9t/git\_intro\_hilt/30c9050817d6317bf9343a812cc7f39203400167/index.html

Now we move through our standard workflow. But what happens to the tree. How would we represent that here?

If we switch back and forth between master and indent, we can see how the different options look easily.

$ git checkout master

$ git checkout indent

Maybe we want to sit with that feature for a while. In the meantime, we've decided we're going to have a bio page. Move back on the master branch (we want to keep the indent branch clean and dealing only with the one feature - indentation.)

$ touch bio.html

And add this content -

https://raw.githubusercontent.com/bmw9t/git\_intro\_hilt/master/bio.html

Standard workflow. We've committed. Update the tree.

So now let's say we've made a choice. We know what we want to indent things. We want that to be the new normal, not a parallel option. We want the new world to contain both a bio page and an indentation page. First get back to the master branch.

$ git checkout master

$ git merge indent

We're telling the computer we want to take the branch indent and merge it back into master, reuniting the two options and pulling the new information from the indent branch. This is pretty common practice - you work on a feature, get it working, and when it's up to snuff you bring it back into the main timeline. How would we represent this on our graph? The log --graph function will give us a hint.

$ git log --graph

Can anyone guess why our indentation style didn't copy affect the bio page? Because it was a change in the style that was only made to the index.html page. How could we get around that? CSS! If these were in a separate file we wouldn't have to worry about it.

Some final tips - branching gets very confusing when the same file is modified in different branches. When you try to merge them back together you get merge conflicts, which cause endless headaches.

So we've covered the basics - we have a workflow, this workflow gives us a history, and we can use version controlling to map out different versions as we work.

~10 for the next section

Now go to the folder you made for your own websites. Turn them into git repositories. Practice making changes to them using the git workflow. When you feel confident with those basics, try making a feature branch for a new feature. Eventually try to merge it back into the main master branch of your project.