Coding Lab 1 `Git` Instructions: `Git` Commits and Checkouts

1 Motivation: What are `Git` and `Github`?

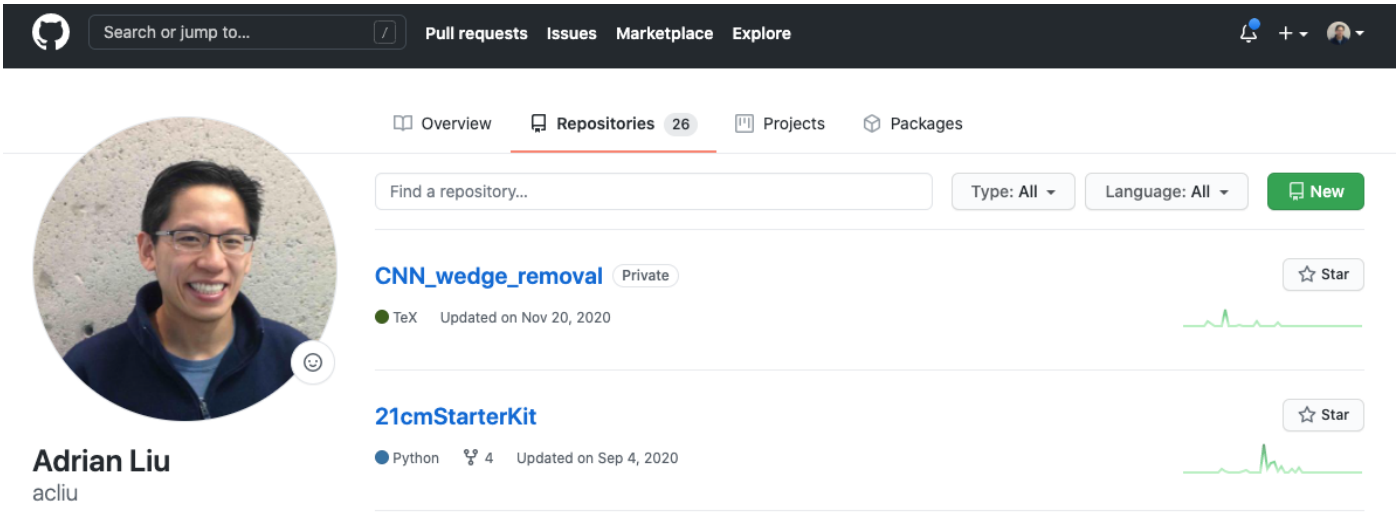
Up until now, you've probably been writing fairly simple programs that are "one and done" codes that you quickly throw together for, say, some lab, and never look at ever again. But as you embark on increasingly complicated projects, it will become much more important to be practice tight *version control*, where you can keep easy track of different stages of your code development, consulting and going back to earlier versions as needed.

In fact, many of us have our own *ad hoc* systems for this. Do you have some folder on your computer with a series of slightly different files named `homework.tex`, `homework_v2.tex`, `homework_final.tex`, `homework_really_final_this_time.tex`? I do! This is an attempt at version control. `Git` solves this problem much more elegantly and powerfully. `Github` is an online platform that allows multiple developers of a code to use `Git` in a collaborative way. If you have some time, here is a fun story illustrating the motivations behind `Git`: http://tom.preston-werner.com/2009/05/19/the-git-parable.html.

2 Getting started

By now, you should have installed `Git` on your computer and have also created an account on the `Github` website (following the Installation Instructions posted on myCourses). If not, please flag down a TA now to help you get all of this done quickly.

Navigate to your account on `Github` and click on the "Repositories" tab. It should look something like this (but probably without existing repositories):



Click the "New" button (in green near the top right corner). This enables us to create a new Github repository where you can share code with others and to collaborate on code in an efficient and elegant manner. Pick a sensible name for your new repo (perhaps `PHYS321_CodingLabs` or something?). Make it a private repo and select the option to add a readme file.

You've now made yourself a nice new Github repo. But it's not very conveniently located, since you will want to write code on your own computer. We're now going to *clone* the repo to your computer. Click the green button labelled "Code" and copy the URL. Now open up a terminal on your computer, navigate to

where you want to put your PHYS 321 materials and type "`git clone {your URL goes here}`". This should download a copy of the repo onto your hard drive.

Type "`git status`". I get something like this:[1]

```
adrian@~/Classes/PHYS321/PHYS321_CodingLabs> git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
adrian@~/Classes/PHYS321/PHYS321_CodingLabs>
```

This isn't so important right now, but notice that because we haven't done anything yet, it says "nothing to commit".

Let's now get organized to get started on Coding Lab 1. We don't want a repo that's completely disorganized, so let's create a folder/directory called `CodingLab1` (or something equivalent). Try to do this on the command line![2] Recall that the relevant commands are in the Installation Instructions handout on myCourses. Download the Coding Lab 1 files from myCourses and unzip them. Put everything in the folder that you created.

3 Your first commit

`Git` helps us perform *version control* by a series of *commits*. Think of a commit as a checkpoint that we can later go back to if we want. (We'll learn how to do that later in this coding lab). Let's now make our first checkpoint/commit.

First, navigate to where you put the Coding Lab 1 files and type "`Git status`" in your Terminal/Git bash program (depending on what operating system you're on). You should see something like this:

```
adrian@~/Teaching/McGill/PHYS321/Winter2021/PHYS321_CodingLabs> git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        CodingLab01_CoordinateSystems.ipynb

nothing added to commit but untracked files present (use "git add" to track)
adrian@~/Teaching/McGill/PHYS321/Winter2021/PHYS321_CodingLabs>
```

Notice how the Jupyter notebook is listed under "Untracked files". This is `Git`'s way of telling us that we haven't yet told it that we'd like to keep track of this Jupyter notebook as we make changes to it. Let's rectify that.

In general, to ask `Git` to keep track of a file for us, we perform the following steps:

1. First, it's always good to check on the status of the `Git` repo to make sure everything is in order. We just did it above, so there's no need right now. But usually I always start by typing "`git status`" .

2. First, type "`git add {location of the file you want to commit}`". If you're in the actual folder where the lab is located, the location of the file is just the filename; otherwise, you'll have to specify the folder.

[1] Actually, yours should say `origin/main` rather than `origin/master`. A few months ago `Github` realized that master vs slave terminology probably wasn't the best name for this.

[2] If you're a Linux user, you probably know how to do this already. On a Mac, open up your `Terminal` app. On a Windows machine, open up the `git bash` program.

3. Type "`git status`" again. It should now tell you that the notebook has been staged for a commit. *It has not been committed yet.* You have simply told `git` that you intend to include this file in your next commit.

4. We'll now do the actual commit. Type `git commit -m` "{`a short description of the changes made in this commit`}". An appropriate commit message might be "First commit of Coding Lab".

Type `git status` again. You should get something like this:

```
adrian@~/Teaching/McGill/PHYS321/Winter2021/PHYS321_CodingLabs> git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
adrian@~/Teaching/McGill/PHYS321/Winter2021/PHYS321_CodingLabs>
```

Notice how it says that you're "ahead of 'origin/main' by 1 commit". This is `Git`'s way of telling you that you've committed an extra set of changes locally on your computer, but those changes haven't been transmitted to the online `Github` repo yet, so we're out of sync. That's ok. We'll deal with that later. For now let's work on the actual Coding Lab.

Open up a new copy of the terminal (or if you are on a windows machine, find the `Anaconda Powershell Prompt` and run that). Navigate to where you put your Jupyter notebook, type "`jupyter notebook`", click the relevant file, and have fun! Follow the instructions in the notebook until it tells you to come back to this PDF.

4 Committing Multiple Files

Welcome back! At this point, you should be at the "Git interlude" section of the Jupyter notebook. We've made a good amount of progress, so it's a sensible time to commit our work again. This is our way of telling `Git` that we're at a good (temporary) stopping point, and we'd like to store a copy of our work in this state so that we can come back to it later if necessary.

Open up a new copy of the terminal and navigate to folder where you have all the coding labs. Type "`Git status`". This time you should get something like this:

```
adrian@~/Classes/PHYS321/PHYS321_CodingLabs> git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   CodingLab01_CoordinateSystems/CodingLab01_CoordinateSystems.ipynb

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        CodingLab01_CoordinateSystems/.ipynb_checkpoints/

no changes added to commit (use "git add" and/or "git commit -a")
adrian@~/Classes/PHYS321/PHYS321_CodingLabs>
```

You'll see that the Jupyter notebook that was originally in my repo has been correctly recognized by `git` as a file that has been modified. Go ahead and follow the instructions from before to make a new commit.

You might be wondering why `Git` is so cumbersome in requiring that we first *stage* a file to be committed (by using the "`git add`" command) before doing a final commit (using the "`git commit`" command). One

of the reasons for this is the ability to add multiple changed files to the same commit. Let's practice this. If you navigate to the "`images`" directory, you'll see a number of image files there that the Jupyter notebook depends on. Make another commit (a *single* commit) that adds all of these images. To do so, you can either issue multiple "`git add`" commands *before* doing a single final "`git commit -m {...}`". Or you can type "`git add {file 1} {file 2} {file 3}`" and then commit as usual.

Let's play with this some more. Type "`touch test.txt`". This is not a `git` command. The `touch` command creates a new (empty) file with a name of your choosing (in this case, `test.txt`). You can verify this by using `ls`. If you now do a "`git status`" again, you will see the new file listed under the list of untracked files. Commit this file.

Now suppose we don't want to keep track of `test.txt` any more. We have two options. If we type "`git rm test.txt`", we will delete the file *and* `git` will stop worrying about it. If we instead type "`git rm --cached test.txt`", the local copy of the file we remain, but `git` will no longer track it. Try out the second option, and then commit the change. Checking `git status` again, you'll see that `test.txt` has returned to the list of untracked files.

If you type "`git log`", you will be able to see a list of all previous commits. You'll see that each commit has what's known as a *git hash*, which is the long string of digits and letters. This is the "ID number" that `git` uses to keep track of the commits. You'll also see the author and date of the commits, along with the commit message. Use the arrow keys to navigate up and down. Press "`q`" when you are done looking around.

Some general tips for committing changes in `git`:

- Commit early and often. Beginners often wait until their whole project is done, and then make one giant commit of everything. This defeats the point of `git`!
- Commits should be small, ideally focused on one task. Let's say you have some function in your code, and you'd like to make two changes to it. For instance, your first change might be to stop hardcoding some parameter values, instead turning them into arguments that are accepted by the function. Your second change might be to change the algorithm for some numerical integral that's performed in the function. These should be two separate commits.
- Commit messages should be short, say, ≤ 3 sentences, although sometimes there are exceptions.
- Commit messages should be clear descriptions of the changes that were made. Beginners sometimes just put the date of their commit. This is useless, because 1) the date and time of each commit are already automatically kept track of by `git`, and 2) it is unlikely that 6 months from now you will say "I'd like to look at what the code was like on January 2nd at 10:26AM". You are much more likely to say "hmm...I don't remember how I did the numerical integral when we were still using the old algorithm. Let me look at the last commit that I made before we implemented the new algorithm."

That's it for now for `git` work! Go back and finish up the Jupyter notebook.

5 Checking out old versions

Congratulations on finishing your first Coding Lab! Save your work, then quit the Jupyter notebook by closing all the relevant windows on your web browser and then pressing `Ctrl-c` twice in the terminal were you launched your Jupyter notebook at the start of the lab. Now do a `git` commit of your notebook.

Suppose that you want to go back and look at an older version of your notebook. We can use the `git checkout` command to do this. Try it yourself. Pull up the `git` log and copy the git hash corresponding to your earlier commit (when you had only completed half of the Jupyter notebook). The syntax is "`git checkout {git hash of the commit goes here}`". When I try this command, I get the following:

```
adrian@~/Classes/PHYS321/PHYS321_CodingLabs> git checkout 87b0ecf593be62b9aa621cd60d5fd431a223ae23
error: The following untracked working tree files would be overwritten by checkout:
        test.txt
Please move or remove them before you switch branches.
Aborting
adrian@~/Classes/PHYS321/PHYS321_CodingLabs>
```

Oh no! What went wrong? Recall from Section 3 that we didn't delete `test.txt`. All we did was to ask `git` to stop keeping track of it for us. What `git` is now saying is that since it's not keeping track of this file, if we revert back to an old version of everything, we'll lose what's in this untracked file. It's suggesting that we either move it somewhere outside of the repository (i.e., some other location on our hard drive) to keep it safe or to just simply remove the file (if we don't care). Since this is just a blank file that we don't care about, go ahead and delete it. (Do it on the command line! Use Google or refer to the Installation Instructions if you don't remember how to do this).

Now try doing the `git checkout` that we were trying to do before (with the correct `git` hash and all that). It should work now. Indeed, if you do a quick `git log`, you should find that your most recent commit isn't there any more. Fire up your Jupyter notebook again. You'll see that your answers to the second half of the questions are gone, as expected. Don't panic, though! `Git` hasn't deleted anything. Close up your Jupyter notebook and type `git checkout main`. This should restore your repo to its most recent commit. Fire up your notebook again to check if you're paranoid about it!

Now, what we just did was useful if we just want to be tourists and look around at an old commit for a little bit. What if, say, we regret what we did on our latest commit and we actually want to go back to an earlier commit? We can accomplish this with the `git revert` command. Let's practice this. Create a file `test2.txt` and write "hello!" in it. Commit this change. Now reopen the text file and change the text to "bye!" Commit this change as well. Let's say that after this last change, we've changed our mind, and want to go back to the original state of this text file (back when it just said "hello!"). What we have to do is to enter the command "`git revert {git hash of commit we want to return to}..{git hash of last commit we want to get rid of}`". This will undo all the changes that happened between these two git hashes. Try using this command to undo our latest commit. When you execute the command, you'll actually bring up a screen that says something along the lines of "Please enter the commit message for your changes". What has happened is that `git` has brought up a text editor within the terminal and they want you to write a brief description of what you're doing (for record keeping). The default message is quite good, so type ":wq" to save and quit (the colon allows you to enter a command, the "w" means write, i.e., save what's written, and "q" means quit). Type "`more test2.txt`" to quickly display what's in the text file. You'll see that it now says "hello!" We've undone the last commit.

If you type `git log`, you'll see that `git revert` accomplishes the reversion by making a new commit. It doesn't rewrite history. An alternative command is "`git reset --hard {git hash goes here}`". This will restore everything to an earlier commit, rewriting history so that it looks like the later commits never happened. Try using this command to get back to the commit prior to the creation of `test2.txt`.

6 Submitting your work

Congratulations on finishing Coding Lab 1! It's now time to submit your work so that we can take a look at it. You can do this by *pushing* your local changes to your repo on `Github`. Type "`git push origin main`". This says to *push* your changes to the "origin" (that's the copy of the repo on `Github`) from your "main" branch (we'll get into what this means next time).

For us to be able to mark your work, we need to have access to your repository. Head back to the `Github` website for your repo and make sure your Jupyter notebook is there. Then, go to "Settings" and click "Manage access". Press the green button labelled "Invite a collaborator" and type in my `Github` username "acliu". This will give me access to your repository.