

Coding Lab 3 Instructions: **Git** Merging

1 Group work

Welcome back! This week we will learn more about **Git**, in particular how to deal with what are known as merge conflicts. This will be the first coding lab where we will practice the sort of coding collaboration that is prevalent in large scientific collaborations. ***If your lab partner(s) are not here today, let us know immediately!***

2 Setting up the repo

In the coding labs so far, I've encouraged you to each maintain your own repos. Today we will work in a slightly different way, in that you will all contribute to the repo of one of your group members.

Select ***one person from your group*** whose repo you plan to work in. This person should navigate to the repo on the **Github** website and add the other group members as collaborators (click "Settings", then "Collaborators") if they haven't been already.

Everyone else in the group now needs to make sure that they have an up-to-date copy of the repo:

- If you have never worked in this repo before, you'll need to clone a copy to your local machine. Do so now! Recall that to do so you want to first navigate to the "Code" tab of the repo on **Github**, click the green "clone or download" tab and copy the URL. Open your terminal, and navigate to *some location that is **not** inside your own `PHYS321_CodingLabs` folder*. (You don't want to create a repo inside a repo). Clone the repo using the command `git clone {insert copied URL here}`.
- If you previously worked on this repo on your local machine, you probably cloned it earlier. Navigate to the repo in your terminal and type `git pull origin main`. This will *pull* all the latest changes Github down to your local machine.

Everyone in the group now needs to clone the repo to their local machine. Navigate to the "Code" tab of the repo, click the green "clone or download" tab and copy the URL. Open your terminal, and navigate to *some location that is **not** inside your `PHYS321_CodingLabs` folder*. (You don't want to create a repo inside a repo). Clone the repo using the command `git clone {insert copied URL here}`.

3 Populating the repo

Let's start putting some files in the repo. ***Someone in the group who was not the person who created the original repo*** should copy the file `hello.py` into the new repo. Commit this file, then push your changes to the repo. (I am being purposely vague about how to do this, because I want you to practice these **git** commands. Refer back to the instructions from previous labs if you like!).

Other members of the group now need to pull down these changes from the **git** repo. Do it by typing `git pull origin main`.

4 Running code on the command line

Today we will be developing code outside of the Jupyter notebook environment. We will write Python scripts and then execute them on the command line. Let's remind ourselves of how this works. Navigate (using the command line) to the directory where `hello.py` lives. Type `python hello.py` and the Python script should execute.

made. A trickier situation occurs when two people work on the *same* part of the code but do different things. Let's see what happens when we do this.

- *One person in the group* should change the line “`print('Hello! My name is Anna. Do you wanna build a snowman?')`” to “`print('Hello! My name is Elsa, and the cold never bothered me anyway.')`”. Commit the result and push to the Github repo.
- *A second person in the group* should change the line “`print('Hello! My name is Anna. Do you wanna build a snowman?')`” to “`print('Hello! My name is Olaf, and I love warm hugs!')`”. Commit the result locally.
- If you are *the third member of the group*, watch and see what happens!

Again, have the *second member of the group* try to push their changes up to Github. Like before, this fails because there are changes in the remote repo that haven't been incorporated. Type “`git pull origin main`” to pull down these changes. This time things aren't quite so successful! You should get a scary looking message that looks like this:

```
57d0675..b0d4fee master -> origin/master
Auto-merging hello.py
CONFLICT (content): Merge conflict in hello.py
Automatic merge failed; fix conflicts and then commit the result.
```

Yikes! This is known as a *merge conflict*. This is `git`'s way of telling us that two users have made simultaneous changes to the code, in a manner that is difficult for `git` to know the “right” way to merge the codes. When this happens, we must *resolve* the merge conflict by stepping in and telling `git` what to do.

Roughly speaking, there are three ways to resolve a merge conflict:

- We can use the command “`git checkout --theirs {insert filename here}`” to tell `git` that you don't want to keep your version of the file, and instead want to go with the other person's changes.
- The opposite approach is “`git checkout --ours {insert filename here}`”, which keeps your changes and discards the other person's changes.¹
- The final option is to review the changes manually and figure out what we want. This is by far the most common way to do things, because usually both coders have written something useful, and we want to surgically pick some things from one person's contributions and other things from the other person's contributions.

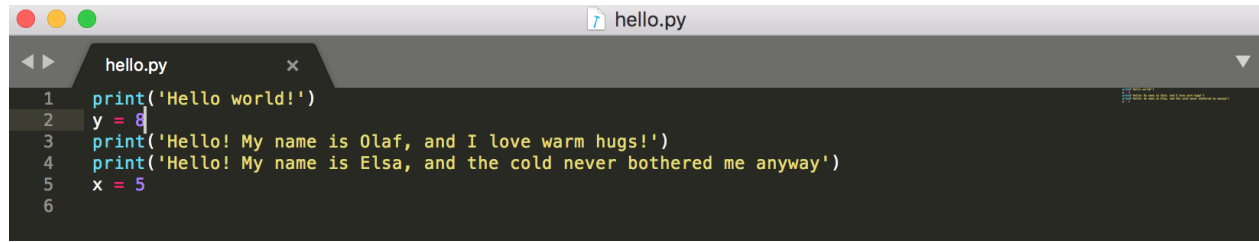
Let's practice the third option. On the computer with the merge conflict, open `hello.py` with a text editor. You should see something like this:

```
1 print('Hello world!')
2 y = 8
3 <<<<<< HEAD
4 print('Hello! My name is Olaf, and I love warm hugs!')
5 =====
6 print('Hello! My name is Elsa, and the cold never bothered me anyway!')
7 >>>>>> b0d4feed17de96d04b2e83540b71369c6825c771
8 x = 5
9
```

Notice how `git` has inserted some markers like `<<<<<<`, `=====`, and `>>>>>>`. This is `git`'s way of showing where the different versions of the code differ. Between `<<<<<<` and `=====` we see the version that we had

¹Of course, as with most things in `git`, when I say “discard” the other person's changes, I don't mean the information is thrown away forever. Remember that `git` is software designed for version control, where we can always go back to an earlier, pre-merge version of the code—an earlier commit—if we want to.

on our local computer, and between ===== and >>>>>> we see the “other” person’s version. What needs to happen now is that we need to replace all of this with the final version that we want. In this case, let’s just delete the lines with the markers, so that we end up with



```
1 print('Hello world!')
2 y = 8
3 print('Hello! My name is Olaf, and I love warm hugs!')
4 print('Hello! My name is Elsa, and the cold never bothered me anyway')
5 x = 5
6
```

In other words, we’re deciding to keep *both* print statements. We didn’t have to do this. We can put whatever we want and remove the markers.

Once we’ve manually fixed the merge conflicts, we can save the text file. Commit the change to tell `git` that the conflict has been resolved. At this point, the workflow is the same as what we had before. Do a push to the remote repo, and then have everyone do a pull to get the latest changes.

7 Git wrap-up

Congratulations! After suffering through `git` these last few coding labs (I know the learning curve can be frustrating), you know how to do the following tasks on `git`:

- Create a `github` repo.
- Clone a repo from `github` onto your local machine.
- Pull new changes from a `github` repo onto your local copy.
- Make `git` commits to keep track of changes.
- Check out old versions of your code from earlier commits.
- Revert back to old versions of your code.
- Push changes to a remote repo.
- Create branches to do code development in a way that does not mess up code that works.
- Merge branches into a main branch when code development is complete.
- Issue a pull request to ask that your development branch be merged into the main branch.
- Fix merge conflicts.

Of course `git` has capabilities beyond what we have learned in the last three coding labs, but the skills that you have developed to this point are sufficient for most day-to-day tasks involving `git`. At this point, it’s mostly just practice until it becomes second nature. It may seem annoying at first, but I strongly recommend using `git` for version control. It’s a good skill to have if you’re ever going to do any collaborative coding. Some people have even taken it to a complete extreme and have planned their weddings using `github`! See <https://readwrite.com/2013/06/12/have-the-nerdiest-wedding-on-the-block-with-github/>.

8 Exercise: Surface Brightness Fluctuation Simulations

This week's coding work is a little bit more open-ended than in previous weeks. Recall in class when we discussed the *surface brightness fluctuation method* for determining distances that if we look at a collection of stars (e.g., in an elliptical galaxies), the picture looks much smoother if the collection is far away, so that there are lots of stars in an individual pixel. This logic can be applied in reverse: by measuring the variance between different pixels, we can infer how far away the galaxy is—the lower the variance, the greater the distance.

This week, you will

- Write some code that simulates this effect. In other words, write code that demonstrates the scaling relation that we discussed in class, where the farther away a galaxy is, the smaller the standard deviation between pixels of the image.
- This week you will **submit one code per group**. Push your code to the repo that you created at the beginning of this coding lab. Remember to add me and your TAs as collaborators!
- In addition to pushing your code, you will also need to push
 1. At least one plot that demonstrates your results. Think about what sort of plot(s) would best illustrate your point.
 2. A short text file with a description of what your code does and how your plot illustrates what you are trying to show. Also include a discussion of how you could make your code more realistic.
 3. A short text file with a description of how you distributed the work between group members. Every member of the group should do an equal amount of work! Be sure to list your names and student IDs in the file.

Once you've discussed the problem with your group and feel that you are ready to start coding, you are strongly encouraged to discuss your strategy with us. A little struggle is excellent, but we don't want groups to spend too much time going down an incorrect path!

As a reminder, here are two different models for collaborating on code:

- Decide on the overall structure of the code and write down a list of functions that need to be written. Divide up the list of functions amongst you. Each person should create a new branch on their `git` repo to indicate which function they are writing. One group member might be working on a branch called `dev.function_1` to develop `function_1`, while another group member might be working on a branch called `dev.function_2` to develop `function_2`. When you think you have something that works, push the branch to the remote `github` repo and issue a pull request for someone else in the group to review (the same way you did on your last coding lab). Only merge the change into the main branch when everyone is satisfied. Note that because different people are writing different functions that will eventually have to interact with each other, it is absolutely crucial to very carefully define the *interfaces*, i.e., the exact inputs and outputs of each function.
- Have two people work in front of the same computer,² practicing what is known as *pair coding*. This is fairly commonly practiced in industry. One person is the *driver*, and does all the typing. This person worries about little details like getting the syntax right. The other person is the *navigator*. This person watches the driver to catch mistakes and also makes suggestions about the overall direction. ***If you decide to do pair coding, you must switch roles frequently.***

Don't forget to push your work to the repo when you are done! Send us a quick message letting us know which group member's repo we should look at for grading.

²Or via screen sharing if you're working remotely.