Git workflows and tutorials

New users to git can often find a steep learning curve, which is often compounded by inexperience using a terminal. These tutorials and workflows provide a simple baseline for using git from the command line and (hopefully) steer users clear of frustrating and confusing errors.

These instructions assume you have already set up an account on Github and have git installed on your local computer.

Creating a remote/local repository pair

To create a new repository we want to make sure the *origin* repository is on Github. All local repositories will track the origin (i.e. they are *downstream* from it). The git commands `push` and `pull` make sense from the perspective of the downstream repository - and since we run commands from the local perspective, the local repository should be downstream.

1) First, go to https://github.com/Your_User_Name, click on your **Repositories** tab, and click on the green **New** button.

2) Fill out the following page - here you can denote the repository as either public or private, and set up a licesne if needed. If the repository is public, you should include a readme file as well to help users understand how to install and use your code.

Now you have created the *origin* or *upstream* master repository. All local repositories will be git clones of this repository.

3) Move to the directory where you want to store your files. Right click and open a terminal in the folder (on mac, you can open a terminal window and drag and drop the file location onto it). The clone command is `git clone https://github.com/Your_User_Name/Repository_Name`.

4) Now the git heirarchy is set up, but the repos are still empty. Use `git add .` to add all files and subfolders in the current directory to the staging area.

5) Use `git commit` to commit the staged changes (added files) to the local repo. You will be asked to provide a description of the changes made - saying "added inital files" is enough.

6) Finally, use `git push` to push the commit *upstream* to the origin. You will be asked for your Github username and password, which are the credentials that allow you to edit the origin - at this point, only you have permissions to do this.

Basic upkeep for an existing local/remote repository pair

If the repositories are set up as described in the previous section, users can completely control both the origin and the local clone with simple terminal commands.

Before working on code, bring up a terminal in the directory of your local repository (right click -> open terminal or, on mac, drag the folder in finder into the terminal window). Use `git pull` to make sure your local repository is up-to-date with the remote origin.

After working on code, bring up a terminal in the directory once again. Use `git add .` to move all new or changed files in the directory to the staging area. You can also use `git add <filename>` to add individual files or folders. Use `git commit` to commit the staging area to the local repository. You will be asked to describe the changes in a short comment - this is important to keep track of what was changed in each iteration. Finally, use `git push` to push the changes to the remote origin that was originally cloned when you set up the local repository. You will be asked for your credentials (Github account) which give you permission to do this.

Using git to work from any other computer

On another computer with git installed (i.e. working from home) you can create another clone of the origin with `git clone https://github.com/Your_User_Name/Repository_Name`. Use the basic upkeep steps in the same way when using the new clone. In this way you can work remotely seamlessly - when you get back to your usual work environment, just run another `git pull` to import the new changes to your usual workspace.

Utilizing branches

Branches are an organizational tool for keeping separate versions of your repository system. The first and default branch is always called "Master". You may want branches for developing features, which you would later merge into the master branch (stable release) without affecting it in the meantime. To create a feature branch in your local repository, use `git checkout -b <branch_name>`. Now you have more options when pushing: `git push -u origin Feature_Branch_Name` will push the new branch to origin, adding upstream tracking with the -u tag. To only push the master branch, you can now use `git push origin master`. This is the same effect as `git push` with only one branch, but now you are specifying to push to the master branch of the origin repository, as there are more options. `git push origin --all` will push all branches to origin - again, add the `-u` tag if there are new branches that origin doesn't have yet.

The point of a branch is to make development easier - you are free to break things while you create new features without affecting the master branch. Ultimately, you want to re-combine the feature branch with master - this is called a merge. We have already used merges before, as `git pull` implicitely merges the origin/master branch into your local master branch. To merge a feature branch locally, use `git checkout master` to swtich back to the perspective of the master branch (the one that will survive the merge) and run `git merge Feature_Branch_Name`.

Collaboration using git

Git can be used for collaboration between multiple people working on the same project. A basic verison of this is described in the "Using git to work from any other computer" section - just imagine that a different person is programming at the other computer. This works seamlessly when there is only one person working on the project at once, but what happens when two people simultaneously work on different things in the repository?

Simultaneous changes

Lets say two people, A and B, both work on the same repository. A makes changes and pushes them to origin, which works as normal, since the changes that B has made are only local so far. But when B tries to push, they will get `error: failed to push some refs to '/path/to/repo.git'`. B is trying to push changes to origin, but from *a version behind*, because of the earier push by A. Origin will not accept changes this way. B has to update their repository, running `git pull --rebase origin master`. This pulls the changes made by A, and uses them to **rebase** B's local repository. If this works without errors, B can simply `git push origin master` to add their changes on top of A's changes.

If A and B are working on the same file, the rebase may fail with `CONFLICT (content): Merge conflict in <some-file>`. Now, it's up to B to create a new version of `<some-file>` incorporating their changes with the version made by A. Git cannot do this automatically! It's a hard enough problem for a human to handle. When B is done, they run `git add <some-file>` to stage the new version and `git rebase --continue` to finish the rebase. B may realize they need help, and can end the process at any point with `git rebase --abort`, returning to where they started. After a successful rebase, B simply runs `git push origin master` to update the origin.

Avoiding conflicting changes

This example shows how tedious it can be to have two people working on the same file. To avoid this, it is good practice to **functionalize** your code as much as possible. This means you should package operations into self-contained functions that are saved as separate files (at least in Matlab). This makes it far less likely that mutliple collaborators edit the same file when developing. It's also better practice in general; it's much easier to debug, and easier for others to read.