**Introduction to Git**

This workshop will help get you started with Git. This tool helps track changes made in project documents such as program files or source codes, effectively versioning them and allowing teams to collaborate via a central repository hub such as Github, Bitbucket, or Gitlab. This workshop covers configuring and initializing a git repository; editing, staging, and committing files; branching and retrieving previous versions of files; and setting up a repository on Github for individual or project team collaboration.

There will be hands-on exercises using git and Github. This workshop will be taught via the command line using Git Bash. All of you received an e-mail yesterday with instructions to download and install the latest version of git, create a Github account, and download the training materials, including the handout. The handout contains the step-by-step guide to the workshop. We will go through each step, which is numbered for easy reference. I will explain each step, demonstrate it, then let you do it. I designed the handout so that you can review what you have learned in this workshop or do the workshop on your own and at your own pace.

If you have any questions, please e-mail me at [foa2@cornell.edu](mailto:foa2@cornell.edu).

**Instructions before workshop**

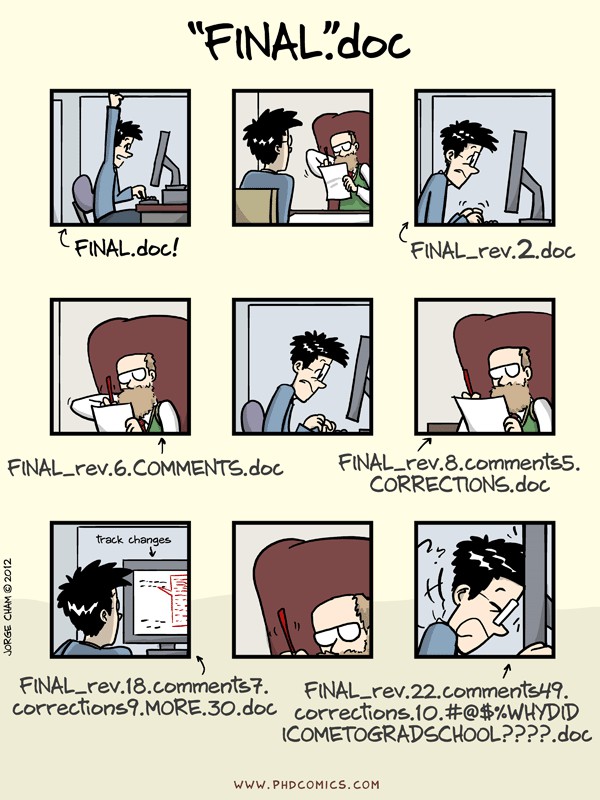
1. **Download and install the latest version of Git** (<https://git-scm.com/downloads>). Accept the default settings when installing the software. This workshop will be using Git for Windows, but should apply to Mac and Linux OS.
2. **Create a Github account** (<https://github.com/>). Take note ofthe e-mail address you used to set up this account. You need this to configure your git account)

1. **Download GitTrain.zip** (<https://cornell.box.com/s/q9s0tu3bunnie4ht607numzk82fwig0u>). Then unzip on your local drive (say C drive). The GitTrain folder will be our project folder and local git repository.

1. **Download the Git\_Workshop\_Handout.docx** (<https://cornell.box.com/s/n88e8nzzqv709tebq1mh516q4z5arz32>). The handout contains the step-by-step guide to this workshop. Best to print this before the workshop or display this on a second monitor.

**About Git**

Why use a git versioning software?



Look at the above comic strip. This comic strip is a great example depicting the need for versioning software. Some of us are probably guilty of this where we have multiple versions of a document, with each version having a different file name. Not only does it increase the number of files on your project folder, but figuring out the latest version could become confusing later. Not only that, if you are collaborating with someone, chances are they will also give the file a different name when they return it to you.

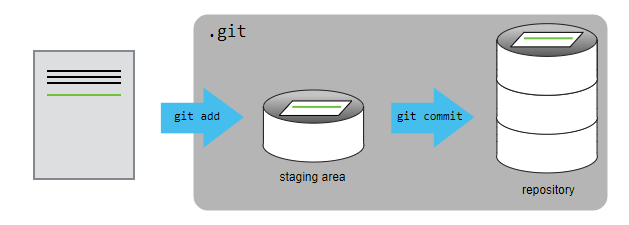
Using git, you and your collaborator can retain the same file name while updating the file. You can also retrieve previous versions of the file should the need arises.

**Features and Importance of git:**

* Secure
* Reliable
* Speed
* Lightweight -- (used right out the box with little to no integration/configuration overhead.)
* Branching -- Enables multiple users to work separately without affecting each other’s work
* Open Source
* Compatibility-- a git is compatible with most of the operating systems in use currently
* Non-linear--allows users from all over the world to perform their operations remotely
* Handles multiple workflows with its speed and wide distribution

Source: <https://www.webku.net/git-are-you-in-a-detached-head-state/#:~:text=A%20detached%20head%20state%20simply%20means%20you%20are,history%2C%20thus%20leading%20to%20an%20old%20project%20release>.

**Starting Git**



Source: https://swcarpentry.github.io/git-novice/04-changes/index.html

Figure 1

**Typical git workflow**

The very first time you use git, you configure it using the command *git config*.

Then, if you haven't done so yet, create a project folder where you will store your project files.   This project folder will become your git repository as well. Once your project folder is ready with all your project files, you make it a git repository using the command *git init*. Then git will create a subfolder named .git within your project folder. It is a hidden folder, but you can see it using the command *ls -a*.

The next step is to tell git which files you need to track for changes using the command *git add*. The *git add*command stages the files to the git staging area*.*One main reason for the staging area is that you will likely track only select files in your project folder. You won't likely track all files in your project, but you can if you want to. You just have to stage all of them.  Another reason for the staging area is that it allows you to change your mind. If you decide not to commit the changes you made to the file; you can remove it from the staging area.

Once you are satisfied with the files you added to the staging area, you save them to the git repository using*git commit*. From here on, git will track for changes all files you've committed. Every time you make a commit, you make a version.

Now let's say you modify a file again; if you want to save the changes you made to that file, you stage it, then commit it.

**HANDS-ON**

**Git syntax**

git *verb options*

For example: git add --all

**Open Git Bash**

(0) Click *Start*>search for *Git Bash*, then hit enter

(This opens the Git Bash terminal or interface that allows users to access the Git Bash command line)

**Configure git**

When we use Git on a new computer for the first time, we need to configure a few things

(1) $ git config --global user.name "Florio Arguillas"

(2) $ git config --global user.email foa2@cornell.edu

*(email must be the same one used to setup the Github account)*

(3) $ git config –-global –l *(to display lists of configuration)*

*Optional:* You may have to **set up your favorite text editor**

Notepad++ (Win, 64-bit install)

(4) $ git config --global core.editor "'c:/program files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"

**Seeking help**

(5) $ git help *(to display list of verbs)*

(6) $ git help <verb> *(to display manual for specific commands)*

**Create a Repository**

We’ll do our work in the C:/GitTrain folder so make sure you change your working directory to it. Use forward slashes:

(7) $ cd C:/GitTrain

(8) $ ls *(to verify you are in the correct directory – ls displays directory contents)*

(9) $ git init *(to initialize and create a repository)*

(10) $ ls

(11) $ ls -a *(displays contents including hidden directories created by git)*

Git saves information about files’ history in the unique .git directory so that the filesystem doesn’t become cluttered (and so that you can’t accidentally edit or delete an old version). If you delete the .git subdirectory, you will lose the project’s history.

(12) $ git status *(checks the status of the project files. Note the untracked files. You pick some or all files you want track)*

**Tracking Changes**

git add – stages files to commit or track

(13) $ git add manifest.txt *(Won’t work because file name is case-sensitive)*

(14) $ git status

(15) $ git add Manifest.txt

(16) $ git status *(git verb is also case-sensitive, but command keyword git is not)*

(17) $ Git add Analysis-Data *(folder name is also case-sensitive – stages contents of Analysis-Data folder)*

(18) $ GIt status

(19) $ GIT add Command-Files *(stages contents of Command-Files folder)*

(20) $ git status

(21) $ git help add

(22) $ git add -A *(add all untracked files)*

(23) $ git status

(24) $ git commit –m “base or initial state of documents” *(the first version)*

Commit message is a description explaining why a particular change was made. Commit messages capture the history of your changes, so other contributors can understand what you’ve done and why. Message has no set character limit, but advice is to limit it to <50.gi

(25) $ git status

**Let’s see tracking in action!**

Now that we’ve committed the files to be tracked for changes, let’s modify some of them.

(26) On the C:\GitTrain\Command-Files folder, open 0-Master.do using a text editor (Notepad) and delete the first \* and save the file.

(27) $ git status *(read the message on the terminal – “modified: Command-Files/O-Master.do”)*

(28) $ git add –all *(stages 0-Master.do. add tells git we are confirming we want to save changes made to this file.)*

(29) $ git commit –m “uncommented the 4th do command of the master file” *(this is the 2nd version of 0-master.do file. Commit tells git to save the file.)*

(30) Let’s edit the same file again. On the C:\GitTrain\Command-Files folder, open

0-Master.do using a text editor (Notepad) and delete the remaining \* and save the file*.*

(31) $ git status *(again shows that 0-Master.do file has been modified)*

(32) $ git add --all *(stages all modified tracked files, in this case 0-Master.do)*

(33) $ git commit –m “uncommented the 5th do command of the master file” *(this is the 3rd version of 0-master.do file)*

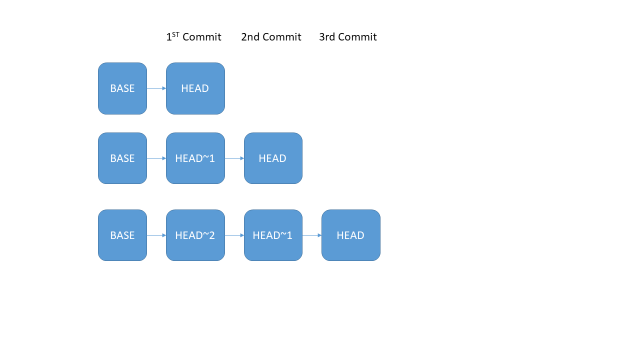
**Viewing differences between versions**

(34) $ git diff HEAD HEAD~1 *(compare latest commit versus previous commit)*

(35) $ git diff HEAD HEAD~2 *(compare latest commit versus 2 commits prior)*

The HEAD pertains to the latest (or most recent) commit on the branch. HEAD is a reference to a commit object. Each branch has its own HEAD. Each time you commit, the most recent commit becomes the HEAD of the branch. Let’s look at Figure 2 below, which shows the status of current and previous HEADs. The first time you commit, the 1st commit is the HEAD. The second time you commit, the 2nd commit becomes the HEAD, and the first commit is referred to as HEAD~1. The third time you commit, the 3rd commit becomes the HEAD and the 2nd commit is designated as HEAD~1, and the 1st commit is designated as HEAD~2. The index number of previous HEADs increases by 1 each time you make a new commit.

Figure 2



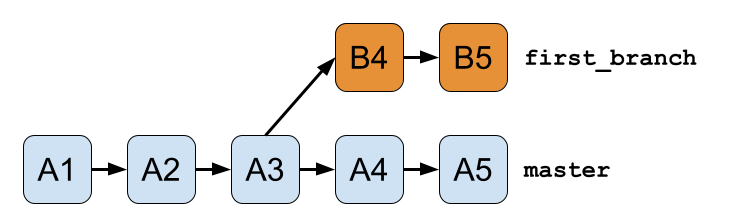
**Branching and retrieving previous versions**

While working in git, the typical state of your git session is that you are currently working in an active branch, like *master*.

***Branching*** is the way to work on different versions of a repository at one time. By default, your repository has one branch named *master*, which is considered the definitive branch. We use branches to experiment and make edits before merging them back to the*master* branch.

When you create a branch off the *master* branch, you’re making a copy, or snapshot, of the *master* as it was then. If someone else made changes to the *master* branch while working on your branch, you could pull in those updates. Figure 3 illustrates this process:

Figure 3



In this figure, we have two branches, *master* and *first\_branch*. *first\_branch* was created as a copy of *master* at commit A3, so commits A1, A2, and A3 are all in the history of both *master* and *first\_branch*. But afterwards, we can continue making changes to each branch without affecting the other branch. So in this example, we’ve made commits A4 and A5 in *master* that are not found in *first\_branch*, and on the other hand, we’ve made commits B4 and B5 in *first\_branch* that are not found in *master.*

When retrieving previous versions, it is best to create a branch from an existing branch (e.g., master) and**w**ork on this new branch to preserve the source branch.

If you want to retrieve a previous version of the file, say the first commit (Designated as HEAD~2), you use the *git checkout <sha1-id of commit>* command, but because you are checking out HEAD~2, the state of your git session becomes a “detached HEAD state.” What does that mean? Since a branch has a HEAD, and you are currently not in a branch, you are in a detached HEAD state. In a way, you are HEADless. If you make changes to files at this state, you might lose those changes when you switch to another branch. If you want to preserve your changes, create a new branch from this state using *git checkout –b <nameofbranch>* then make the necessary changes to the file.

Source: <https://guides.github.com/activities/hello-world/>

Let’s check out the first version of our file (the first commit, which is designated as HEAD~2). But first, let’s create a branch from master to preserve the master branch.

(36) $ git checkout –b <nameofbranch>

*(let’s use* *first\_branch* *as the name of our new branch; -b causes a new branch to be created that has all the revision history of the current active branch, in this case master)*

(37) $ git log *(to identify the sha1-ID of the version to be retrieved)*

git log lists all commits made to a repository in reverse chronological order. The listing for each commit includes:

-The commit’s complete identifier (which starts with the same characters as the abbreviated identifier printed by the git commit command earlier)

-The commit’s author

-The log message returned during the commit

https://swcarpentry.github.io/git-novice/04-changes/index.html

Tip: log may produce an extended output, and it may seem you cannot go back to the main console window. Just press Q to go back to the main console.

(38) $ git checkout <sha1> (to checkout a particular commit, where sha1 is commit’s unique ID; choose the 2nd commit – uncommented 4th do file)

You will see the following note from Git, which we will discuss in a moment: *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*

Note: Check the Command-Files folder. The 5th do command should have an asterisk again because that was its state during the second commit.

(39) $ git status

Head detached at <sha1>

Let’s talk more about this “Head detached” status, which you will recall Git notified us about during Step 39. A “detached head” is a state wherein a particular commit gets checked out instead of a branch. Problem with this is that this does not belong to any branch. If you make changes and commit them they can easily get lost once you checkout a different version or branch.

You can start a new branch at this stage to make that checked out version belong to a branch. This would you to retain changes made to this branch. So let’s create a new branch at this stage.

(40a) $ git checkout –b <nameofbranch> (let’s call this new branch *second\_branch*)

Or just discard the changes you made and go back to the original branch (master)

(40b) $ git checkout master

**Merging branches**

The *master* branch is considered the definitive branch, while we use other branches to experiment and make edits safely without affecting *master*. But what if we decide that we like the changes made in some branch, and want to now include them in *master*? For this, we can use ***merging***. Merging lets us pull changes from one branch into another. Think back to the example illustrated in Figure 3. In Figure 3, none of the changes made in commits B4 and B5 affect *master.* If we want those changes to be part of *master*, we can merge *first\_branch* into *master*, and that will allow B4 and B5 to be included in the commit history of *master*. Let’s try this ourselves:

(41) $ git branch *(display list of branches – active branch with asterisk)*

(42) $ git checkout first\_branch

(43) On the C:\GitTrain\Command-Files folder, open the 1-import-wdi.do file using a text editor (Notepad), insert Hello in row 6, and save the code.

(44) $ git add --all

(45) $git commit -m "inserted Hello in row 6 of 1-import-wdi.do"

(46) $ git log *(this branch now has 4 commits, compared to 3 for master)*

*Checkout to the desired branch (in this case we want to merge first\_branch to master)*

(47) $ git checkout master *(to make master the active branch)*

(48) $ git merge first\_branch *(merge first\_branch to master)*

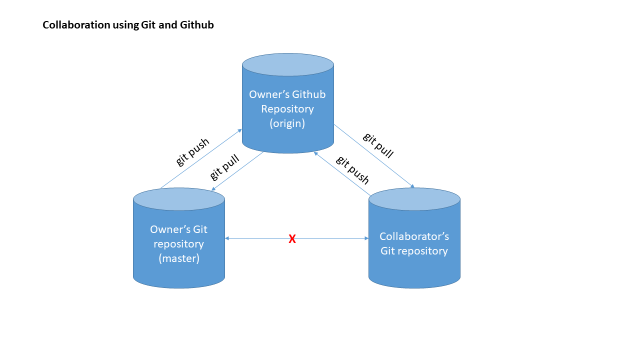
(49) $ git log *(master now has 4 commits, 1 came from first\_branch)*

**Forks** are often used in open source development on GitHub. A fork, also known as branch, is a copy of a repository that you manage. Forks let you make changes to a project without affecting the original repository. You can fetch updates from the original repository with pull requests. You can submit changes to the original repository provide you have push (write) access to it.

<https://docs.github.com/en/free-pro-team@latest/github/collaborating-with-issues-and-pull-requests/working-with-forks>

**Collaboration Using Git and Github**

**Figure 3.**



**Typical workflow for collaboration**

Each collaborator's local computer is likely not directly connected, and therefore, they need a central hub to act as a conduit between the two.

So the owner (or PI) who wishes to invite collaborators must set up an account at a git repository hosting service such as Github, Gitlab, or Bitbucket. In our case, GitHub plays the cloud-hosted, online, Git repository that teams can use to store code centrally.

On Github, the owner will create a new repository. It is empty initially, but it will provide a URL that the owner needs to sync their local git repository to their remote repository on Github.

The owner will link the two repositories using *git remote add origin <url>*

Then pushes the contents of the local repository to the remote repository using *git push origin master*.

Then on Github, the owner invites the collaborator.

The collaborator will clone the repository on their local computer using *git clone <url> <destination>*

The collaborator can now make changes to the files and stage them using *git add*.

Then commits the changes with *git commit* and uploads the changes to Github using *git push origin master*. This sync's the remote repository on Github with the collaborator's repository.

The owner then updates or syncs their local repository using *git pull origin master*. This process of pushing, syncing, and pulling will go on until project completion.

**Git versus Github**

Just to clarify the difference between git and github. “Git is a distributed version control tool that can manage a project's source code history, while GitHub is a cloud based platform built around the Git tool. Git is a tool a user installs locally on their computer, while GitHub is an online service that stores code pushed to it from computers running the Git tool. The key difference between Git and GitHub is that Git is an open-source tool users install locally to manage source code, while GitHub is an online service to which Git users can connect and upload or download resources.”(Mckenzie, 2018) [Git vs. GitHub: What is the difference between them? (theserverside.com)](https://www.theserverside.com/video/Git-vs-GitHub-What-is-the-difference-between-them#:~:text=Git%20is%20a%20tool%20a%20developer%20installs%20locally,tool%20developers%20install%20locally%20to%20manage%20source%20code,)

**Let’s do some hands-on!**

1. Login to your Github account
2. Create a new repository. Click *New* (Repositories)
3. Assign a Repository name (let’s use GitTrain, but it doesn’t have to be the same as your <zoommtg://cornell.zoom.us/sso?token=c1hEp_rRtN8T1vmMRRNMMUd6B6085JW_zqJ0Yhf8FY_1h82VKT_izPulshTcNQtaPlzLG-Ye6VM7fXFnasrvcOhXXvqROAwiQxYeX0sqLIRv4CvgBPY-6zl5gE1NAZPt0pwI7JGTWv6Wki_-ulLSBhjHpAW5lPOyBoYRzIo60W6G_e4gB5d6OT9J7UFDcqcbWdT6_6xyElslm6m0leXLpUTDgMz2TMjlMVitevOBWpnjxR0ReNnx_TYoVXxzg1UjClONWMj5ktqorhw.aIL6e7ZR3P_IsKBr&code_challenge=xLaoC1oGQk2/i+geS15Ol8MerZltYpzm8OgPRbeDQoA=>local repository)
4. (Optional) Add a description
5. Select *Private*
6. Skip the other boxes because we will import an existing repository
7. Click *Create a Repository*

Github displays a page with URL and some info on how to configure your local repository.

At this stage you now have two repositories, one local (your computer), the other remote (Github)

1. Connect the two repositories. Click HTTPS and copy the URL written on the text box next to it.
2. Go to your local computer, and on Git Bash, navigate to your local GitTrain repository, and run this command:

$ git remote add origin <paste\_github\_repository\_URL>

**origin** is a local name used to refer to the remote repository.

1. Check that the command worked:

$ git remote -v

origin https://github.com/xxxx/GitTrain.git (fetch)

origin https://github.com/xxxx/GitTrain.git (push)

1. Push all the changes from the local repository to the Github repository.,

$ git push origin master *(push copies changes from local to remote repository)*

At this stage the local and remote repositories are in sync (identical)

1. We can pull changes from the remote repository to the local one as well:

$ git pull origin master (pull copies changes from a remote to a local repository)

Already up to date since we did not make changes in Github

1. On the Github repository, click the Code tab, navigate to the Command\_Files folder, and click on *2-import-pew.do* file. This opens up the file.
2. Click the Edit tool (pencil icon) and *insert Hello in row 6* of the file
3. Scroll all the way down and on the text box under commit changes, type *Inserted Hello in row 6 of 2-import-pew.do*
4. Accept the default Commit directory to the master branch
5. Click *Commit changes*
6. Go to your local computer, and on Git Bash, navigate to your local GitTrain repository, and run this command:

$ git pull origin master (this will sync remote and local repositories)

**Sharing your repository with collaborator.**

*Owner:*

1. On Github, click on *Settings>Manage Access>Invite a collaborator*, and type the username, full name, or e-mail of your collaborator on the search box.
2. Once the collaborator is found, add them to the repository.
3. Inform the collaborator that they will receive an e-mail invitation from Github to access the repository and that they should accept that invitation.

*Collaborator:* Will clone the owner’s repository to their local computer.

1. Collaborator logs in to their own Github account.
2. Collaborator opens the e-mail and accept the Github invitation to access owner’s repository
3. On the Github repository, click on Code, then click on the Code pulldown (green color) and copy the URL found under Clone.
4. On the local computer, opens Git bash, types the following:

$ git clone <paste\_github\_repository\_URL> <local\_destination\_folder>

The local destination folder is where the collaborator wants to put the git repository. It doesn’t have to be an existing folder. If it does not exist, git will create it.

For example: Let’s say the destination folder on the local computer of the collaborator is C:/GitTrainProj

$ git clone https://github.com/foa2/GitTrain.git C:/GitTrainProj

Note: Collaborator may be prompted to enter their Github username and password once or twice. A login failure message might appear, but this gets resolve by entering your username and password when prompted.

1. Then collaborates navigates to the destination folder:

$ cd C:/GitTrainProj

1. Then collaborator makes changes to some files. For example:

On the C:\GitTrainProj\Command-Files folder, open the 3-processing.dofile using a text editor (Notepad), insert *Hello* in row 6, and save the code.

1. $ git add Command-Files/3-processing.do
2. $ git commit –m “inserted hello in row 6 of 3-processing.do”
3. $ git push origin master

*(this syncs collaborator and the Github repository, but owner repository still not up to date)*

Note: Collaborator may be prompted to enter their Github username and password once or twice.

*Owner and Collaborator:*

1. *Optional:*  Owner and Collaborator may verify the Github repository to see if it has been updated.

*Owner:*

1. $ git pull origin master (to sync owner repository with the Github repository. By this time all three repositories--owner, Github, and collaborator-- are in sync).

**Other useful links:**

<https://dev.to/nichartley/whats-a-git-tree-5149>

<https://www.freecodecamp.org/news/how-to-delete-a-git-branch-both-locally-and-remotely/>

<https://swcarpentry.github.io/git-novice/> (highly recommended Git workshop module)

Gienow, Michelle (2018) <https://thenewstack.io/dont-mess-with-the-master-working-with-branches-in-git-and-github/>