1. git --version
2. git config user.name -------- to check if you have any username
3. git config --global user.name “Eklavya Bishnoi” ---------- this is to define the username in global to show your name if you made any edits.
4. git config user.name
5. git config --global user.email [eklavya.bishnoi@gmail.com](mailto:eklavya.bishnoi@gmail.com) ----- use this to assign the email
6. Terminal Comands
   1. LS – list the contents of the folder in the current directory
      1. ls folder\_name --- it will show what is in this folder
      2. ls -a --- it lists all file such as hidden files too
   2. Open . – if I type “ open . “ then it will open the current folder in finder
      1. Open folder\_name – it open the folder in the finder
   3. Cd – change directory is used to enter into a folder
      1. Cd .. ---- cd with two dots makes us move out of the folder by one level
   4. Mkdir – command to make a directory
      1. Mkdir plants
   5. Pwd ---- this command prints the current working directory
   6. Clear --- clears the terminal screen
   7. Touch ---- touch filename and it will create a file with that name in the folder where we are in the terminal
      1. You can create multiple files with different extensions you just have to use touch and then filename\_1, filename\_2, filename\_3
         1. Touch filename\_1.ipynb filename\_2.py filename\_3.txt filename4.pdf
      2. Use Touch folder\_name/file\_name.extension to create that file in a folder
   8. Rm ---- to remove any file use the rm command – this deletes the file permanently
      1. Rm file\_name
      2. Rm -rf -- deletes a folder -🡪 rm -rf folder\_name
7. Git Commands
   1. Git status --- it gives information on the current status of a git repository and its contents
   2. Git init --- used to create a new git repository. Before we can do anything git-related we must initialize a repo first. This is something you do once per project initialize the repo in the top-level folder containing your project
   3. Always use Gitstatus first to confirm that you are not in a respository. If you use Git init in an already existing repository then you would confuse git down the line. There should be only one git init per repository/repo.
8. Committing
   1. It is the most important Git feature
   2. Working Directory – it is the directory that I am working in.
   3. Staging Area --- you stage the changes that you want to make using the git add command and then once you have staged them you will make the changes to the repository using the git commit.
      1. Adding ---- “ git add filename1 filename2” ---- separate files with spaces to add multiple at once.
      2. Committing – use “git commit -m “Message here in quotes” “ – always use git commit -m “My message” ----- while you are committing new changes.

Stage using “git add” and then put it in the repository using “git commit”

* 1. Repository --- actual git repository the .git folder where you are making the changes or commits
  2. Git Commit -m “my message’
     1. We use the “git commit” command to actually commit changes from the staging area
     2. When making a commit, we need to provide a commit message that summarized the changes and work snapshotted in the commit
  3. Git commit -a -m “ Message here” ---- this does both work of adding all the unsaved changes to the staging area and then committing the changes to the branch.
  4. Git Log : It retrieves the information for the logs of the commits for a repository
  5. Git add . ---- it stages all the files with the changes for the commit. If there are multiple changes to the files tracked by git then it will add all the files for the commit.

1. Code . --- this command opens up the current folder in VS Code
2. The Git Docs--- talks about git documentation
   1. Keep your commits atomic
      1. When possible , a commit should encompass a single geatuere, change or fix. In other words, try to keep each commit focused on. Asingle thing. This makes it much easier to undo or rollback changes later on. It also makes your code or project easier to review.
   2. Use Present-Tense Imperative Style ---- Describe your changes in imperative mood, e.g. “make xyzzy do frotz” instead of [This patch] makes xyzzy do frotzz or “I changed xxyzzy to do frotz”, as if you are giving order to the codebase to change its behavior
3. Closer look at the Git Log Command
   1. Git log --abbrev-commit
   2. Git log – -oneline
4. GitKraken – GUI to commit using a graphic user interface.
5. AMMEND COMMITS
   1. Suppose you just made a commit and then realized you forgot to include a file! Or, maybe you made a typo in the commit message that you want to correct.
      1. Rather than making a brand new separate commit you can “redo” the previous commit using the –amend option
         1. Git commit -m “some commit”
         2. Git add forgotten\_file
         3. Git commit --amend
      2. NOTE – The Git Commit amend only amends the latest commit not the previous commits
6. IGNORING FILES
   1. We can tell git which files and directories to ignore in a given repostiroy , using a .gitignore file. This is useful for files you NEVER want to commit including:
      1. Secrtes, API Keys, Credentials, etc
      2. Operationg System Files (.DS\_Store on Mac)
      3. Log Files
      4. Dependencies and Packages
   2. .gitignore – Create a file called .gitignore in the root of a repository. Inside the file, we can write patterns to tell Git which files and folders to ignore:
      1. .DS\_Store will ignore files name .DS\_Store
      2. folderName / will ignore an entire directory
      3. \*.log will ignore any files with the .log extension
7. Gitignore.io file that needs to be added
   1. gitignore should be added to the base folder
   2. Use .gitignore.io to copy paste general files that needs to be out of io section.
8. BRANCHING
   1. Think of branches as alternative timelines for a project.
   2. They enable us to create separate contexts where we can try new things, or even work on multiple ideas in parallel.
   3. If we make changes on one branc, they do not impact the other branches (unless we merge the changes)
   4. In Git, we are always working on a branch, The default branch name is master.
      1. It doesn’t do anything special or have fancy powers. It is just like any other branch.
      2. Many people designate the. Master branch as their source of truth or the official branch for their codebase, but that is left to you to decide.
      3. From Git’s perspective, the master branch is just like any other branch. It does not have to hold the master copy of your project.
      4. HEAD
         1. Head is simply a pointer that refers to the current “location” in your repository. It points to a patruclar branch reference.
         2. So far, HEAD always points to the latest commit you made on the master branch, but soon we will see that we can move around and HEAD will change.
   5. GIT BRANCH
      1. Use git branch to view your existing branches. The default branch in every git repo is mater, though you can configure this.
      2. Look for the which indicates the branch you are currently on.
   6. Creating Branches
      1. Use git branch <branch-name> to make a new branch based upon the current HEAD.
         1. This just creates the branch. It does not switch you to that branch (the HEAD stays the same)
   7. Switching Branches – GIT SWITCH
      1. Git switch <branch\_name> to switch it.
   8. GIT CHECKOUT
      1. Git checkout <branch-name>
      2. Historically we used git checkout <branch-name> to switch branches. This still works.
      3. The checkout command doeas a million additional things, so the decision was made to add a standalone switch command which much simpler.
      4. You will see older tutorial and docs using checkout rather than switch
   9. CREATING and SWITCHING BRANCHES
      1. Git switch -c <branch\_name> ----- use this line of code to create a new branch and switch to it all in one go.
      2. Remember -c as short for create
9. DELETING AND RENAMING BRANCHES
   1. Delete --🡪 git branch –delete or git branch -d
   2. Rename -🡪 git branch -m ------ to rename we have to be in the branch.
      1. Git branch -m new\_branch\_name
10. MERGING
    1. Branching makes it super easy to work withing self-contained contexts, but often we want to incorporate changes from one branch into another! We can do this using the Git merge command
       1. We merge branches and not specific commits
       2. We always merge to the current HEAD branch
    2. Merging Made Easy
       1. Switch to or checkout the branch you want to merge the changes into (the receiving branch)
       2. Use the git merge command to merge changes from a specific branch into the current branch
          1. Git switch master
          2. Git merge bugfix
             1. This is called a fastforward merge when we are tying to catch up the other branch with the work done on recent branch. In this case there is no work on the master branch done since the split
          3. Things to remember in fast forward commit --- first of all move to the branch in which the commit needs to be made and then use the git merge branch\_name command to merge the commits of the branch with the main branch in which we want to merge the branches
11. Git DIFF Command
    1. We can use the git diff command to view changes between commits, branches, files, our working directory and more.
    2. We often use git diff alongside commands like git status and git log, to get a better picture of a repository and how it has changed over time.
    3. Without additional options, git diff lists all the changes in our working directory that not staged for the next commit.
    4. Compares the difference between staging area and the working directory
    5. Git Head
       1. Shows changes in the working directory with the staging area
    6. GIT Diff Head
       1. List all changes in the working tree since your last commit
    7. Git diff --staged or git diff --cached
       1. Both these will list the changes between the staging area and our last commit
       2. Show me what will be included in my commit if run git commit right now
    8. Diff – ing specific files
       1. Git diff HEAD [filename]
       2. Git diff --staged [filename]
    9. Difference between branches
       1. Git diff branch1..branch2
          1. The order of the branches matter when we are checking the results. The changes from branch 1 will be displayed first with a – sign and then changes from branch 2 will be displayed using the + sign.
    10. Comparing two commits
        1. Git diff commit1..commit2
           1. Compares the changes between commit 1 and commit 2
           2. The commit1 should correspond to the alpha numberic number of the commit. So the command could look like
              1. Git diff 4a9da7b..0b75516
    11. Git Stash
        1. Git provides an easy way of stashing these uncommitted changes so that we can return to them later, without having to make unnecessary commits.
        2. Git stash:
           1. is super useful command that helps you save changes that you are not yet ready to commit. You can stash changes and then come back to them later.
           2. Running git stash will take all uncommitted changes (staged and unstaged) and stash them, reverting the changes in your working copy.
        3. Git stash pop :
           1. to remove the most recently stashed changes in your stash and re-apply them to your working copy
        4. Git Stash apply :
           1. You can use git stash apply to apply whatever is stashed away, without removing it from the stash. This can be useful if you want to apply stashed changes to multiple branches.
        5. Stashing multiple times:
           1. You can add multiple stashed onto the stack of stashed. They will all be stashed in the order you added them.
           2. Git stash list:
              1. You can view the stash list of the Git

To apply which stash do you want to apply you can use the following syntax

Git stash apply stash@{2} --- this applies the stashed changes on list at number 2

* + 1. Dropping Stashed
       1. To delete a particular stash, you can use git stash drop <stash-id> : git stash drop stash@{0} ---- drops the stashed change at 0
    2. Git Stash Clear : it clear out all the stashes, run git stash clear
  1. Undoing stuff and time travelling
     1. Git Restore
     2. GIT Reset
        1. Git Reset acutally moves the branch pointer backwards, eliminating commits. So those commits are gone.
     3. GIT Revert :
        1. Git revert <commit-hash>
        2. It is similar to git rest in that they both undo changes but they accomplish it in different ways.
        3. Git revert instead creates a brand new commit which reverses/undos the changes from a commit. Because it results in a new commit, you will be prompted to enter a commit message.

GIT HUB

Github is a hosting platform for git repositories. You can put your own git repos on GitHub and access them from anywhere and share them with people around the world.

Beyond hosting repos, GitHub also provides additional collaboration features that are not native to Git (but are super useful). Basically, GitHub helps people share and collaborate on repos.

Cloning

Git clone <url> ----- Git will revtrieve all the files accosicated with the repository and will copy them to your local machine. In addition, Git initializes a new repository on your machine, givng you access to the full Git History of the cloned project.

So far we have created our own Git respositories from Scratch, but oftern we want to get a local copy of an existing repository instead.

To do this, we can clone a remote resporitory hosted on Git hub and similar websites. All we need is a URL. That we can tell Git to clone for Use

Git config user.email ----- this tells you which email git is using

SSH Key

You need to be authenticated on Github to do certain operations, like pushing up code from your local machine. Your terminal will prompt you every single time for your Github email and passoerd, unless….

You generate and configure an SSH key!once configures your can connect to Git hub without having to supply. Your username/password

Remote

Before we can push anything up to Github, we need to tell Git about our remote repository on Github. We need to setup a destination to push up to.

In Git, we refer to these “destinations” as remotes. Each remote is simply a USL where a hosted repository lives

Git remote -v

To view any existing remotes for your repository, we can run git remote or git remote -v (verbose for more info)

This just displays a list of remotes. If you haven’t added any remotes yet, you wont see anything.

Adding a new remote

Git remote add <name> <url>

A remote is really two things: a URL and a label. To add anew remote, we need to provide both to Git

Origin

* Origin is a conventional Git remote name, but it is not at all special. Its just a name for a URL.
* When we clone a Github repo, the default remote name setup for us in called origin, You can change it. Most of people leavel it.
  + 1. Git remote rename <old> <new> ----- this is to rename a remote
    2. Git remote remove <name> --- this gets rid of the remote all together

5. Pushing

1. Now that we have a remote setup, lets pus some work up to GitHub! To do this we need to use the git push command
2. We need to specify the remote we want to push up to AND the specific local branch we want to push up to that remote.
3. Git push <remote> <branch>
4. Git push origin main ---- tells git to push up the main branch to our origin remote