*** WELCOME TO THE GIT CHEAT SHEET ***

AUTHOR: ESHAN KALP TRIVEDI (Github: EshanTrivedi21)
REFERENCE: GIT GITHUB BOOTCAMP - COLT STEELE (UDEMY)
DESCRIPTION: Your very own Git AND Github Cheat Sheet having basic Windows Terminal Cheats as well!

*** GO ON FROM HERE, ALL THE BEST ***

TERMINAL BASICS:

>> ls (it is the LIST command which LISTS all the contents of a Directory)

>> ls foldername (LOCATES to a directory and then LISTS the contents)

>> pwd (PRINT WORKING DIRECTORY)

>> cd C:\User\Eshan\foldername (CHANGES and HOPS onto the respective Working Directory)

>> cd .. (HOPS onto the Parent Directory of the Working Directory)

>> clear (CLEARS the used Terminal)

>> q (QUITS out of a entered command)

>> mkdir foldername (CREATES an Empty directory inside the Working Directory)

>> rm filename (DELETES the file)

>> rm -rf foldername (DELETES the directory)

>> start . (OPENS the File Explorer to the ROOT Directory)

SETTING GIT USER NAME AND EMAIL:

>> git config user.name (to CHECK if git User-Name is set)

>> git config user.email (to CHECK if git User-Email is set)

>> git config --global user.name "Eshan Trivedi" (To SET or CHANGE git User-Name)

>> git config --global user.email eshan.trivedi.9@gmail.com (To SET or CHANGE git User-Email)

>> git config --global core.editor "code --wait" (to SET or CHANGE default code editor as VsCode)

>> git config --global --edit (to CHANGE git User-Name and git User-Email directly in the config file using vim)

>> escape + :wq (To exit vim)

GETTING GIT HELP

>> git command --help (Use for more details and examples on any of the below (or above) commands e.g. push, pull etc.)

CREATING A GIT REPOSITORY:

STEP 1: >> git init (INITIALIZES an empty repository)

STEP 2: >> git status (to CHECK the status of a repository, a .git directory is created, all git history is deleted if .git is deletec)

STAGING FILE/FILES OF THE REPOSITORY:

(to keep a track of modifications or changes)

>> git add filename.txt (STAGES the file)

>> git rm --cached filename.txt (UN-STAGES the file)

>> git add --all OR >> git all . OR >> git add . (STAGES all files in the repository)

COMMITING A COMMIT:

(come on who doesnt know about commits)

>> git commit -m "commit message" (COMMITS the STAGED files with a commit message)

>> git commit -a -m "commit message" (SKIPS the Staging part and directly COMMITS)

>> git log (LOGS all the commits done to the repository)

>> git log --oneline (LOG commits in a single line)

>> git commit --ammend (AMMENDS the previous commit)

>> git diff (LOGS the changes)

EVERYTHING ABOUT GIT IGNORE:

(git doesnt consider the files/folders which are in .gitignore)

STEP 1: create a .gitignore file

STEP 2: add files or folders inside the file to ignore, now the files are are untracked by github and wont be staged or commited

GIT BRANCHING:

(if you are trying something out but do on want to play with the main branch)

>> git branch (LOGS all the branches of the Working Repository with an * in the Current Head or GO to .git\refs\heads)

>> git branch branchname (CREATES a new branch but the branch is not yet SWITCHED)

>> git switch branchname (SWITCHES the Branch from one to another or CHANGING HEAD, but the current changes need to be STASHED or COMMITED)

>> git switch -c newbranch (CREATES a new branch and then SWITCHES to the branches)

>> git checkout -b newbranch (CREATES a new branch and then SWITCHES to the branch)

>> git branch -D newbranch (DELETES newbranch, but you cant be Headed on the Branch you want to delete)

>> git branch -M finalbranch (CHANGES the name of the Branch you are Headed on)

MERGING BRANCHES:

FAST FORWARD MERGE (NO CHANGES done on the Master Branch)

STEP 1: >> git switch master (SWITCH the HEAD first to the first branch)

STEP 2: >> git merge newbranch (MERGES newbranch into master with HEAD on master)

NOT ALL MERGES ARE FAST FORWARD MERGES

1. WITHOUT MERGE CONFLICTS

>> git switch master

>> git merge -m "mergering message" (CREATES a new commit unlike fastforwarding merges)

2. MERGE CONFLICTS

(only if conflict message occurs)

STEP 1: OPEN UP files having merge conflicts

STEP 2: REMOVE the conflicts

OPTION 1: ACCEPT INCOMING CHANGES

OPTION 2: ACCEPT CURRENT CHANGES

OPTION 3: ACCEPT BOTH CHANGES

OPTION 4: COMPARE CHANGES

STEP 3: REMOVE the conflict markers

STEP 4: STAGE and COMMIT the changes

STASHING IN GIT:

(needed when switching branches but the changes arent commit ready but by not stashing, the changes will behave wierdly)

1. CHANGES WILL EITHER COME IN THE DESTINATION BRANCHE

2. GIT WONT ALLOW SWITCHING IF THERE ARE CONFLICTS

(hence to prevent this staging is important, its like a save but doesnt show up anywhere unless popped)

>> git stash (STASHES the changes)

>> git stash pop (UN STASHES the changes, use it when you resume your work)

>> git stash apply (APPLY stashed changes into another or the same branch)

IF WORKING WITH MULTIPLE STASHES

>> git stash list (LOGS all the stashes)

>> git stash apply stash@{1} (STASHES the changes in the Stash index 1)

>> git stash drop stash@{1} (DELETES the stash, p.s. applying the stash doesnt delete it)

>> git stash clear (CLEARS the whole stash list)

TIME TRAVELLING WITH GIT:

1. TO JUST CHECK WHAT THE REPOSITORY LOOKED LIKE IN THE COMMIT ID 604a39a

>> git checkout 604a39a (DETACHES HEAD and attaches it to the commit with the commit id 604a39a, this is not normal because HEAD is meant to map a whole branch and not a specific commit)

OR

>> git checkout HEAD~1 (DETACHES HEAD and ATTACHES it to the previous commit)

>> git switch master (RE-ATTACHES HEAD and now the head properly maps onto the master branch)

2. TO CREATE AND WORK WITH A NEW BRANCH AT COMMIT ID 604a39a

>> git checkout 604a39a

>> git switch -c "new branch" (now the head is perfect where it should have been)

3. To RESTORE ALL CHANGES TO TRACKED FILES

>> git reset origin/main --hard

4. TO RESTORE CHANGES OF A PARTICULAR FILE TO THE LAST COMMIT

>> git checkout HEAD filename.txt

OR

>> git restore filename.txt

5. TO RESTORE CHANGES OF A PARTICULAR FILE TO THE SECOND-LAST COMMIT

>> git restore --source HEAD~1 filename.txt

6. UNSTAGE A FILE

>> git restore --unstaged filename.txt

7. RESETING THE REPOSITORY TO A PARTICULAR COMMIT

>> git reset 604a39a (NOTE: it resets the head to the commit id 604a39a, but it doesnt delete the changes, it like there is no commit made after commit id 604a39a)

>> git reset --hard 604a39a (LOOSES the commit as well as LOOSES the contents of the commit)

8. REVERTING THE REPOSITORY TO A PARTICULAR COMMIT

>> git revert 604a39a (REVERTS the changes in that particular commit and CREATES a new commit after reverting changes p.s. this helps while collaboration)

CREATING GITHUB RESPOSITORIES:

1. BUILD A NEW REPOSITORYAND START WORKING FROM SCRATCH (BY REMOTING)

STEP 1: CREATE A NEW REPOSITORY ON YOUR GITHUB WEBSITE AND COPY THE URL

STEP 2: >> git init (CREATES an empty git repository)

STEP 3: WRITE YOUR PIECE OF CODE

STEP 4: >> git commit -a -m "first commit" (a commit is needed to push any files to github)

STEP 5: >> git remote add origin <copied url> (CREATES a new REMOTE DESTINATION for the github repository)

>> git remote -v (LOGS out the REMOTE URL if any)

STEP 6: >> git push -u origin branchname (PUSHES the last commited code to github p.s. the -u is like a setting the origin remote and master branch as a default so that we can use just >> git push in future)

1. BUILD A NEW REPOSITORY AND START WORKING FROM SCRATCH (BY CLONING)

STEP 1: CREATE A NEW REPOSITORY ON YOUR GITHUB WEBSITE AND COPY THE URL

STEP 2: >> git clone <copied url> (CLONES as well as automatically sets the REMOTE DESTINATION)

STEP 3: WRITE YOUR PIECE OF CODE

STEP 4: >> git commit -a -m "first commit" (a commit is needed to push any files to github)

STEP 5: >> git push -u origin branchname (PUSHES the last commited code to github p.s. the -u is like a setting the origin remote and master branch as a default so that we can use just >> git push in future)

3. CONNECT YOUR PREEXISTING GIT REPOSITORY TO A NEW GITHUB REPOSITORY

STEP 1: CREATE A NEW REPOSITORY ON YOUR GITHUB WEBSITE AND COPY THE URL

STEP 2: >> git remote add origin <copied url> (CREATES a new REMOTE DESTINATION for the github repository)

>> git remote -v (LOGS out the REMOTE URL if any)

STEP 3: >> git push -u origin branchname (PUSHES the last commited code to github p.s. the -u is like a setting the origin remote and master branch as a default so that we can use just >> git push in future)

THE origin/master THEORY:

origin/master IS CALLED AS A REMOTE TRACKING BRANCH, IT IS A REMOTE BRANCH THAT REPRESENTS OUR LOCAL BRANCH

>> git branch -r (LOGS the remote tracking branch)

>> git checkout origin/master (to check out the remote branch code: DETACHES HEAD onto the remote tracking HEAD, in a case where the local branch is ahead of the remote branch and not up to date, push to make it up to date)

NOW IF YOU CLONE A REPOSITORY WITH MULTIPLE BRANCHES AND RUN >> git branch THEN TECHNICALLY ALL THE BRANCHES SHOULD HAVE BEEN LOGGED, BUT THIS IS NOT THE CASE ONLY MASTER BRANCH IS LOGGED, THIS IS BECAUSE BY DEFAULT ONLY LOCAL MASTER BRANCH IS CONNECTED TO THE REMOTE BRANCH BUT OTHERS NEED TO BE CONNECTED IN ORDER TO WORK WITH THEM AND THE SIMPLEST WAY TO DO SO IS:

>> git branch -r (to check all the branches available)

>> git switch branchname (this automatically CONNECTS the two branches and we can freely work on them now)

>> git branch (now this correctly LOGS all the connected branches)

FETCHING AND PULLING:

FETCHING ALLOWS TO GET CHANGES FROM THE GITHUB REMOTE REPOSITORY TO OUR LOCAL GIT REPOSITORY BUT DOESN'T CHANGE INTO THE WORKING DIRECTORY

>> git fetch (To receive the new commits)

>> git fetch origin or >> git fetch origin branchname (CREATES a new branch having the changes but this doesnt interfere in the working directory, the origin/master now heads on to this new branch and the master branch would be one branch behind the origin/master)

>> git checkout origin/master (to check out the remote branch code: DETACHES HEAD onto the remote tracking HEAD, in a case where the local branch is ahead of the remote branch and not up to date, push to make it up to date)

PULLING INTERFERES IN THE WORKING DIRECTORY (REGULAR PULL = FETCH + MERGE)

>> git pull origin or >> git push origin branchname (PULLS the changes and merges the changes with the branch you want to OR default)

SOLVE MERGE CONFLICTS IF ANY

FORCE PUSH AND PULL :

>> git push origin <your_branch_name> -f (This will delete your previous commit(s) and push your current one.
f signifies force)

>> git pull --rebase=interactive or git pull --rebase=i (This is nothing but fetch + rebase. This will invoke rebase in interactive mode where you can choose how to apply each individual commit
that isn't in the history you are rebasing on .)

REFER TIME TRAVELLING WITH GIT for more git force pulling commands

GITHUB COLLABORATION:

CENTRALIZED WORKFLOW: EVERY BODY WORKS ON THE SAME MAIN BRANCH AND PUSH AND PULL IN THE MAIN BRANCH ONLY.

FEATURE BRANCH WORKFLOW: EVERY FEATURE IS PUSHED UPON ANOTHER INDEPENDANT BRANCH TO MINIMIZE MESS

GITHUB PULL REQUEST WORKFLOW: Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.

GITHUB FORKS AND CLONE WORKFLOW: Forking nd then opening a pull request to the owner of the main repository.

GIT REBASING:

REBASES OR SEPARATES THE HISTORY OF THE NEWBRANCH FROM THE MASTER BRANCH

>> git switch newbranch

>> git rebase master (REBASES or MERGES master into newbranch)

CONFLICTS WHILE REBASING

STEP 1: Resolve the CONFLICTS MANUALLY

STEP 2: >> git add .

STEP 3: >> git rebase --continue

INTERACTIVE REBASING

>> git rebase -i HEAD~n (OPENS up the code editor where you can play with the nth last commits and above)

reword (CHANGES the name of a specific commit)

fixup (COMBINES the changes of that commit to the previous commit and then deletes it)

drop (DELETES the commit and the commit changes as if they were never written)

GIT TAGS:

TAGS A COMMIT AND MARKS IT WITH THE TAG (DENOTES THE IMPORTANCE OF A COMMIT)

Semanting Versioning Format is widely used for tags and releases (v1.0.1)

>> git tag -l (LOGS all the tags)

>> git tag -l "v17" (LOGS tag name v17)

>> git tag -l "v17*" (LOGS tag names that start with v17..)

>> git tag -l "*beta*" (LOGS tags which include beta in their name)

TO CHECKOUT THE CODE AT THE TAG, USED CHECKOUT

>> git diff v16.0.1 v16.1.1 (LOGS the changes between both the versions)

>> git tag tagname (CREATES a lightweight tag)

>> git tag -a tagname (CREATES an annotated tag)

>> git show annotation (LOGS the message of the git tag "annotated")

>> git tag -a tagname 604a39a (CREATES a tag at commit number 604a39a)

>> git tag -d tagname (DELETES the tag)

>> git push origin tagname (TR5ANSFERS tags to remote servers)

GITHUB DESKTOP:

Interact with GitHub using a GUI instead of the command line or a web browser

1.Installing and authenticating

STEP 1: Download GitHub Desktop for Windows using this [URL](https://desktop.github.com/).

STEP 2: Click on “File” on the navbar, go down to “Options,” choose “Accounts,” and get authentication.

2.Contributing to projects with GitHub Desktop

STEP 1: You can create a new repository by selecting the File menu and clicking New repository.

STEP 2: You can add a repository from your local computer by selecting the File menu and clicking Add Local Repository.

STEP 3: You can clone a repository from GitHub by selecting the File menu and clicking Clone Repository.

STEP 4: You can use GitHub Desktop to create a branch of a project.

STEP 5: After you make changes to a branch, you can review them in GitHub Desktop and make a commit to keep track of your changes.

STEP 6: You can use GitHub Desktop to create issues or pull requests to collaborate on projects with other people.

STEP 7: When you make changes to your local repositories or when other people make changes to the remote repositories, you will need to sync your local copy of the project with the remote repository.

*** YOU DID IT, SIT BACK AND BE PROUD OF YOURSELF ***