Git/GitHub Notes and Comments

* git init
  + creates a new repo, either an empty one or one with files already in it
  + --bare flag creates a repo without a working directory, usually central repos are created as --bare since it is a storage facility rather than a work environment
  + git init: transform current directory into a git repository
  + git init <directory>: create an empty git repository in the specified directory
  + git init --bare <directory>: initialize an empty git repository, but omit the working directory
* git clone
  + copies and existing git repository, is the most common way for users to obtain a development copy
  + git clone <repo>: clone the repository located at <repo> onto the local machine
  + git clone <repo> <directory>: clone the repository located at <repo> into the folder called <directory> on the local machine
* git config
  + git config --global user.name <name>: define author name
  + git config --global user.email <email>: define author email
  + git config --system core.editor <editor>: define the text editor used
  + git config --global alias <alias-name> <git-command>: create a git shortcut
  + git config --global --edit: open the global config file for manual editing
* git add
  + tells git you want to include updates to a file in the next commit
  + the git staging area is a buffer between the working directory and project history
  + you can make all the edits you want, and then use git add to commit the changes piece-by-piece so it is easy to track bugs and revert changes
  + git add <file>: stage all changes in <file> for next commit
  + git add <directory>: stage all changes in <directory> for next commit
  + git add –p: begin an interactive staging session that lets you choose portions of a file to add to the next commit
* git commit
  + commits a stages snapshot to the projects history including a commit message
  + snapshots are always committee to the local repo
  + each developer’s local directory is a buffer to the working directory
  + git records the content of each file in every commit
  + git commit: commit the staged snapshot
  + git commit –m “<message>”: commit the staged snapshot with the message
  + git commit –a: commit a snapshot of all changes in the working directory
* git status
  + two type of untracked files, files that haven’t been committed, and compiled binaries like .pyc, .obj, .exe
  + git allows you to use .gitignore to ignore those files by redirecting the pathway
  + git status: lists which files are staged, unstaged, and untracked
* git log
  + allows the user to see the entire commit history
  + it is used to find specific commits, so the user can perform specific actions of those commits
  + git log: display the entire commit history using the default formatting
  + git log –n <limit>: limit the number of commits by <limit>
  + git log --oneline: condense each commit to one line
  + git log --stat: include which files were altered and the relative number of lines that were added or deleted from each of them
  + git log –p: displays the patch representing each commit, this shows the full diff of each commit
  + git log --author=”<pattern>”: search for commits by a particular author
  + git log --grep=”<pattern>”: search for commits with a commit message that matches <pattern>
  + git log <since>..<until>: shows only commits that occur between <since> and <until>
  + git log <file>: only displays commits that include the specified file
  + git log --graph --decorate --oneline: --graph will draw a text based graph of the commits, --decorate adds the names of branches or tags of the commits that are shown, --oneline shows the commit information on a single line
* git checkout
  + this command serves three purposes: checking out files, commits, and branches
  + checking out a commit allows the user to view an old state of a project without any repercussions on the working directory
  + checking out a files allows the use to view an old state of that file without touching the working directory
  + git checkout is a safe way to load any of these safe copies of a project onto a development machine
  + when the user checks out a previous commit, the head becomes detaches and points to the previous commit
  + the user can recommit an old version of a snapshot to revert back
  + git checkout master: to get back to the master branch
  + git checkout <commit> <file>: checkout a previous version of a file
  + git checkout <commit>: update all files in the working directory to match the specified commit
* git revert
  + this action generates a new commit that undoes the changes made in a previous commit, so in the end you are still at the master branch
  + this prevents git from losing history, which is important for collaboration
  + git revert does not “revert” back to the previous state of the project
  + for example, if a bug was introduced at a certain commit, git revert can be used to fix it and commit a new snapshot
  + revert is a safe option, especially for commits already published to a shared or common repository
  + git revert <commit>: generate a new commit that undoes all of the changes introduced in <commit>
* git reset
  + git reset is the dangerous method, with the potential for the user to lose all of their work
  + should only be used to undo local changes, never any snapshots that have been shared with the developer
  + without the --hard flag, git reset is useful to clean up a repository by unstaging changes or uncommitting a series of commits in order to rebuild them
  + the --hard flag should only be used when an experiment has gone wrong and the user needs a clean slate to work with
  + git reset is designed to undo local changes
  + never use git reset <commit> when any snapshots after <commit> have been published to a public repository
  + if other team members try to sync up with your repository, it will appear to them that they are missing a chunk of the project history
  + as soon as you add new commits, git will think that your local history has diverged from origin/master
  + git reset --hard HEAD^2 moves the current branch backward by 2 commits, this should only be done on unpublished commits
  + git reset <file>: remove the specified file from the staging area, but leave the working directory unchanged
  + git reset: reset the staging area to match the most recent commit, but leave the working directory unchanged
  + git reset --hard: in addition to unstaging changes, the --hard flag tells git to overwrite all changes in the working directory
  + git reset <commit>: move the current branch tip toward <commit>, reset the staging area, but leave the working directory alone, lets the user recommit the project history using cleaner, more atomic snapshots
  + git reset --hard <commit>: move the current branch tip toward <commit> and reset both the staging area and the working directory to match
* git clean
  + this command removes untracked files from your working directory, but the command is not undoable
  + git clean is often used in conjunction with git reset –hard, allowing the user to return the working directory to the exact state of a particular commit
  + it is often used to remove the .o and .exe binaries generated by compilers
  + if a user is lost, using git reset --hard and git clean --df, will make the staging area look like the most recent commit
  + git clean –n: shows you which files are going to be removed without actually doing it
  + git clean –f: remove untracked files from the current directory
  + git clean –f <path>: remove untracked files, but limit the operation to a specified path
  + git clean –df: remove untracked files and untracked directories from the current directory
  + git clean –xf: remove untracked files from the current directory as well as any files that git ignores
* git commit --amend
  + this command lets the user combine staged changes into the previous commit, so you only have one snapshot, useful for fixing premature commits
  + it also lets you edit the commit message without changing its snapshot
  + amending doesn’t just alter the most recent commit, but replaces it, so that to git it looks like a brand new commit
  + never amend commits that have been pushed to a public repository, because it has the same result as resetting a public snapshot, the other developers’ project history will vanish
  + the --no-edit flag will allow you to make the amendment to your commit without changing to commit message
  + git commit --amend: combine the staged changes with the previous commit and replace the previous commit with the resulting snapshot
* git rebase
  + rebasing is just moving a branch from one commit to another
  + git accomplishes this by creating new commits and applying them to the specified base
  + merging directly results in a 3-way merge and a merge commit
  + rebasing and then merging results in a fast-forward merge and a perfectly linear history
  + rebasing is a common way to integrate upstream changes into your local repository, rather than using git merge, which results in a superfluous merge commit every time
  + never rebase commits that have been pushed to a public repository, because it would erase part of your project history
  + git rebase <base>: rebase the current branch onto base, which can be any kind of commit reference (an ID, branch name, tag, or relative reference to HEAD)
* git rebase –i
  + this begins and interactive rebasing session, which gives you the opportunity to alter individual commits
  + it gives the developer complete control over the project history, so you can initially commit a “messy” history and then later combine commit, delete obsolete ones, and make sure the project history looks in order before submitting the official one
  + interactive rebasing keeps a project’s history clean and meaningful
  + the squash command allows the developer to combine two commits
  + git rebase –I <base>: this opens an editor where you can enter commands on how individual commits will be rebased
* git reflog
  + reflog contains information about old states of branches and allows the developer to go back to that state if necessary
  + every time the current head gets updated a new entry is added to the reflog
  + reflog only tracks movements, and with git reset it provides a safety net if changes have been committed to the users local repository
  + git reflog: show the reflog for the local repository
  + git reflog --relative-date: show the reflog with relative date information
* git remote
  + this command allows you to create, view, and delete connections between repositories
  + remote connections serve not as real-time access to repositories, but more so as references using convenient names rather than longer URLs
  + git remote: list the connections you have to other repositories