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
  + checking out a branch updates the files in the working directory to match the version stored in that branch and tells git to store all new commit on that branch
  + checking out branches is not a read-only operation, changes are saved in that branch
  + git checkout allows you to work on multiple features in the same repository
  + HEAD is git’s way of referring to the current snapshot, when the user checks out a commit git switches to a “detached HEAD” state
  + If you develop in a detached HEAD state, you have no way of merging your feature, so always develop on a branch
  + 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 checkout <existing-branch>: checkout an already created specified branch
  + git checkout –b <new-branch>: create and checkout <new-branch>
* 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
  + when a user clones a repo with git clone, they automatically create a remote connection called origin pointing back to the cloned repo
  + HTTP is an easy way to allow anonymous, read-only access to a repository
  + SSH is used for read-write access, but the user would need a valid SSH account on the host machine
  + Having a connection to your teammates’ repos allows for collaboration outside the central repo
  + git remote: list the connections you have to other repositories
  + git remote –v: list the connections including the URL you have to other repositories
  + git remote add <name> <url>: create a new connection to a remote repository using name as a shortcut for url
  + git remote rm <name>: remove the connection to the remote repository called <name>
  + git remote rename <old-name> <new-name>: rename a remote connection from <old-name> to <new-name>
* git fetch
  + this command allows you to import commits as remote branches instead of normal branches; this gives the user a chance to review changes before integrating them into the project
  + remote branches represent commits from someone else’s repo, but you can checkout a remote branch just like local one
  + git fetch is useful for synchronizing your local repository with a central repository
  + git fetch <remote>: fetch all of the branches from this repository, including all of the commits and files stored in the repo
  + git fetch <remote> <branch>: fetch only the specified branch
  + git branch –r: view your remote branches
* git pull
  + git pull merges upstream changes into your local repository, but instead of using git fetch and git merge, git pull is just one command
  + git pull is an easy way to synchronize your local repository with upstream changes
  + the --rebase option can be used to ensure a linear history by preventing unnecessary merge commits
  + git pull <remote>: fetch the specified remote’s copy of the current branch and merge it into the local copy
  + git pull --rebase <remote>: same command as above, but uses rebase instead of merge to integrate remote branch into the local one
  + git config --global branch.autosetuprebase always: after running this command all git pull commands will use git rebase instead of git merge
* git push
  + pushing is how you transfer local commits to a remote repository, this had the potential to overwrite changes
  + git push is essentially the same as running git merge master from inside the remote
  + git prevents the user from pushing when it results in a non-fast-forward merge, so if the remote history has diverged from your local history, you need to pull the remote and merge it into your local one
  + otherwise the --force flag overrides this and makes the remote branch identical to your local one, you must be sure that none of your teammates have pulled those commits before using --force
  + you should only push to repositories made with --bare, since it’s important to never push to another developer’s repository
  + always make sure that local master is up-to-date before using git push
  + git push <remote> <branch>: push the specified branch to <remote>, along with all the commits and files; git won’t let you push when it results in a non-fast-forward merge
  + git push <remote> --force: the –force flag allows you to push even if results in a non-fast-forward merge
  + git push <remote> --all: push all of your local branches to <remote>
  + git push <remote> --tags: tags are not automatically pushed when the user pushes a branch or uses the --all option, so this sends the users local tags to the remote
* Pull requests
  + Pull requests make it easier for developers to collaborate by providing an interface to discuss proposed changes
  + A pull request is a notification that a developer has completed a feature and is also a dedicated forum to discuss that feature and if further ramifications are needed
  + A pull request is you requesting another developer to pull a branch from your repository to their repository
  + So you need the source repo, the source branch, the destination repo, and destination branch in order to complete the request
  + Pull requests work with the Feature Branch, Gitflow, and Forking Workflow, but not with Centralized Workflow because it does not have two distinct repositories or two distinct branches
  + Feature Branch Workflow uses a shared repository for collaboration letting developers create features on isolated branches. Developers open pull requests in order to initiate a discussion and then merge their branch into master
  + Developers can also file pull requests in order to get help from others while implementing a feature
  + Gitflow Workflow defines a strict branching model designed around the project release, features are generally merged into the develop branch, while release and hotfix branches are merged into the both develop and master branches
  + In Forking Workflow, a developer pushes a completed feature to their own public repository and then files a pull request. This allows developers to collaborate inside the pull request and when they’re done, file another pull request and merge the feature into the master
* git branch
  + a branch is a way to request a new working directory, staging area, and project history
  + when the user wants to add a new feature or fix a bug, you should make a new branch to encapsulate your changes
  + this allows the developer to keep their main branch clean with little to no unstable code
  + git stores a branch as a reference to a commit, so a branch represents the tip of a series of commits as a pointer
  + in essence, merges in the project history are a joining of two independent commit histories
  + once fully merged, a branch is no longer necessary and should be deleted
  + git branch: list all of the branches in your repository
  + git branch <branch>: create a new branch called <branch>. This does not checkout the new branch
  + git branch –d <branch>: delete the specified branch, but this safe operation prevents you from deleting the branch if it has unmerged changes
  + git branch –D <branch>: force delete the specified branch, even if it has unmerged changes
  + git branch –m <branch>: rename the current branch to <branch>
* git merge
  + git merge allows the user to take independent branches and integrate them into one single branch
  + all of the commands presented below merge into the current branch, allowing the target branch to remain unaffected
  + a fast-forward merge can occur when there is a linear path from the current branch tip to the target branch, so all git has to do is move the current branch tip up to the target branch tip
  + if the branches have diverged, there is not a linear path to the target branch, so git has to combine them using a 3-way merge using a dedicated merge commit to tie the two histories together
  + developers like to use fast-forward merges for small feature or bug fixes, while reserving 3-way merges for the integration of longer-running features
  + in 3-way merges, if two branches you’re trying to merge both changed the same part of the same file, git stops the merge so you can manually resolve the conflict
  + running git status allows the developer to see which file needs to be resolved
  + if your feature branch did not contain many commits, you’re probably better off rebasing it onto master and doing a fast-forward merge
  + git merge <branch>: merge the specified branch into the current branch
  + git merge --no-ff <branch>: merge the specified branch into the current branch, but always create a merge commit; this is useful for documenting all of the merges that occur in your repository
* Centralized Workflow
  + Centralized Workflow uses a central repository to serve as the single point of entry for all changes to the project
  + developers start by cloning the central repository, so in their own local copies they can edit files and commit changes locally, and when ready to publish they can push their local master branch to the central repository
  + if a developers local commit diverge from the central repository, git will refuse to push their changes since it overwrites official commits
  + in order to get around this, developers need to fetch the updated central commits and rebase their changes on top of them, resulting in a linear history
  + just like in merging, if local changes conflict with upstream commits, git will pause rebasing and allow the developer to resolve this manually
  + rebasing works by transferring local commits to the master branch one at a time, so you catch merge commits on a commit by commit basis
  + this makes it easier to find out where bugs were introduced
  + git pull --rebase origin master: this tells git to move all of the commits to the tip of the master branch after syncing all of the changes from the central repo
  + git rebase --abort: allows the developer to go back to where they were before rebasing
* Feature Branch Workflow
  + adding feature branches to a development process is an easy way to encourage collaboration and streamline communication between developers through pull requests
  + feature development should never occur in the master branch, so the master branch will never contain broken code
  + feature branches should be pushed to the central repository in order to share code, backing up local commits without touching the master branch
  + once someone finished a feature, they file a pull request which allows other developers to review and discuss changes before they are merged with master
  + pull requests are also useful if a developer needs help implementing a feature
  + the –u flag adds branches as a remote tracking branch to the central repository
* Gitflow Workflow
  + this branching model defines strict roles for team members based around the project release
  + it uses individual branches for preparing, maintaining, and recording releases, assigning strict roles to different branches and defining how they should interact
  + instead of having 1 branch to record history, the master branch stores the official release history and the develop branch serves as an integration branch for features
  + each new feature should be its own branch, but instead uses develop as the parent branch, so who a feature is completed it merges back into develop
  + feature branches should never interact directly with master
  + once develop has enough features for a release, you fork a release branch off of develop only bug fixes, documentation generation, and other release-oriented tasks should go off in this branch
  + the only work done in the release branch should be to clean up the release, test everything, update the documentation
  + release branches act as a buffer between feature development(develop) and public releases(master)
  + once it’s ready to ships, release should be merged with both develop and master
  + maintenance or hotfix branches are used to quickly patch production releases, and are forked directly off of master and as soon as the fix is complete, it should be merged into both master and develop
  + have dedicated lines of development for each task lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle
* Forking Workflow
  + this workflow model is fundamentally different than the rest, since it gives every developer a server-side repository, so each contributor has two git repositories: a private local one, and a public server-side one
  + the main advantage of this workflow is that developers push to their own server-side repositories, and only the project maintainer can push to the official repository
  + so the developers do not have write access to the official codebase
  + this makes it an ideal workflow for open source projects
  + when a developer wants to start working on a project, they fork the official repo and create their own personal public copy of it on the server
  + while no other developers are allowed to push to their repo, developers are allowed to pull from it
  + developers then clone a copy to work on as their local repository, and push to their own public repository and file a pull request to the main/central repository
  + the maintainer then pulls, merges, and pushed onto the public central repo and all other developers should pull from that to stay up to date
  + while most workflows need 1 remote, forking requires two
* Merging vs. Rebasing
  + merging is nice because it in a non-destructive feature, so the existing branches are not changed in any way
  + but every time you incorporate any upstream changes and extraneous merge commit gets added to the project history
  + rebase rewrites the project history by creating brand new commits for each commit in the original branch history
  + this results in a much cleaner project history and eliminates the unnecessary merge commits
  + however rebasing project history can be potentially catastrophic for your collaboration workflow and it deletes all context from the merge commit
  + interactive rebasing allow you to alter commits are they are rebased, usually to clean up a messy history before it is merged onto master
  + by using the commands you can changes commit messages, condense two commits into one, and eliminate insignificant commits
  + the golden rule of git rebase is to never use it on public branches
  + since rebasing results in brand new commits, git will think that your master branch’s history has diverged from everybody else’s
  + the only way to sync them back together is to merge them back together, resulting in an extra merge commit and two sets of commits that contain the same changes
  + if anyone else is looking at that particular branch, do not rebase, but otherwise you are safe to rewrite history as much as you want
  + force pushing overwrites the remote master branch to match the rebased one from your repository and this can make things very confusing for your teammates
  + the only times you should be force pushing is when you’ve performed a local cleanup after you’ve pushed a private feature to a remote repository
  + by periodically incorporating interactive rebase into your workflow, you can make sure each commit is meaningful and focused
  + when rebasing you have two options for the new base: the feature’s parent branch or an earlier commit in your feature
  + the latter is nice when you only have to fix up the last few commits
  + git merge-base: find the original base of a feature branch
  + using git merge-base and git rebase you can completely rewrite an entire feature
  + if you are collaborating with other developers, you are not allowed to rewrite history
  + its fine to rebase onto a remote branch, when you’re collaborating with other developers on a feature
  + avoid using git rebase after creating a pull request, since the branch is now a public branch, so clean up the project history using rebase before submitting your pull request
  + when incorporating upstream changes into a feature branch, if you rebase before you merge, you are guaranteed the merge will be fast-forwarded, resulting in a perfectly linear history
* Reset, Checkout, and Revert
  + Commit-Level Operations
    - git reset and checkout operate on whole commits when you don’t include a file path as a parameter
    - resetting is a way to move the tip of a branch to a different commit, which can be used to remove commits from the current branch
    - dangling branches get deleted the next time git performs a garbage collection
    - --soft: the staged snapshot and working directory are not altered in any way only commits
    - --mixed: the staged snapshot is updated to match the specified commit, but the working directory is not
    - --hard: the staged snapshot and working directory are both changed to match the specified commit
    - git checkout lets you switch between branches on the commit-level, but doesn’t move branches around
    - git forces you to commit any changes in the working directory that would be lost when changing branches during the checkout operation
    - git checkout also lets you arbitrarily checkout previous commits, but it does put you in a detached head state, where it is unwise to add new commits
    - git revert undoes a commit by creating a new commit, which is considered a safe way to undo changes, since it has no way of rewriting the commit history
    - git revert should be used to undo changes on a public branch, while git revert should be used to make changes on a private branch
  + File-Level Operations
    - git reset and checkout can have file paths as parameters, which dramatically alters their behavior and limits their operations to a single file
    - when working on the file-level git reset updates the staged snapshot to match the version in the specified commit
    - the --soft, --hard, and –mixed flags do not have any effect on the file-level of git, because the staged snapshot is always updated, but the working directory is never updated
    - checking out a file updates the working directory instead of the stage, and does not move HEAD
    - if you stage and commit a checked out file, this has the effect of reverting back to the old version and removes all of the other changes to that file
    - you can also discard unstaged changes by using git checkout with head as the commit reference
* Advanced Git Log
  + Formatting Log Output
    - --oneline: condenses each commit to one line and only displays commit id and first line of the commit message
    - --decorate: makes git log display all of the references (branches, tags, etc.) that point to each commit
    - --stat: displays the number of insertions and deletions to each file altered by each commit (modifying a line is 1 insertion and 1 deletion)
    - -p: the entire diff log or all the changes introduced by each commit
    - shortlog: intended for release announcements, it groups each commit by author and displays the first line of each commit message
    - --graph: draws an ASCII graph depicting the branch structure of the commit history
    - --pretty=format:”<string>”: this lets you display each commit using printf-style placeholders like %cn for committer name or %h for abbreviated commit hash
  + Filtering the Commit History
    - -<n>: you can limit the amount of git log’s output by putting a number for n
    - --after=/--before=: if you are looking for a commit between a specific time frame you can use these flags to look for it, also you can pass in references like “yesterday” or “1 week ago”
    - --author=: when you’re looking for a commit made a particular individual, can also do “John\|Mary” to find more than 1 person at a time
    - --grep=: filters by commit message, and work like the --author flag
    - by file: if you’re only interesting in seeing changes in a file, use -- <filename> to find all of the commit
    - -S”<string>”: the pickaxe searches for all commits that add or delete a particular line of code, this shows you when a line was copied or moved to another file
    - -G”<regex>”: this searches for a regular expression instead of a string
    - <since>..<until>: you can pass a range of commits to show only the commits in that range, it’s very useful when you use branch ranges in order to find differences between branches
    - --no-merges: prevents git log from showing merge commits
    - --merges: this is only merge commits
* Git Hooks
  + these are scripts that run automatically every time a particular event occur in git and let you customize git’s internal behavior and trigger customizable actions at key points
  + common uses for git hooks include encouraging a commit policy, altering the project environment based on the state of the repository, and implementing continuous integration workflows
  + hooks can reside in either local or server-side repositories and they are executed only in response to actions in that repository
  + by changing the shebang line, git hooks allows the developer to write scripts in whatever language they are most comfortable in
  + local hooks are not copied over when you run git clone, so a simple solution to this is to store your hooks in the actual project directory
  + local hooks only affect the repository in which they reside in
  + pre- hooks let you alter the action that’s about to take place, while post- hooks are used only for notifications
  + pre-commit: this script is run every time you run git commit before git asks the developer for a commit message or generates a commit object, while the hook is used to inspect the snapshot that is about to be committed
    - no arguments are passed into the pre-commit script and exiting with a non-zero status aborts the commit
    - git diff-index: find errors like trailing whitespace, lines with only whitespace, a space followed by a tab inside the initial indent of a line
    - git rev-parse --verify: checks if argument HEAD is a valid reference, if it isn’t then an empty commit is stored in the against variable instead
    - git diff-index --cached --check: compares a commit against the index, while warning the developer if the changes introduced any whitespace errors
  + prepare commit message: this script is called after the pre-commit hook to populate the text editor with a commit message, this is a good place to alter the automatically generated commits messages for squashed or merged commits
    - 3 arguments can be passed to the script: the name of the temp file that contains the message, the type of commit (message, template, commit, squash), the SHA1 hash of the relevant commit
    - exiting with a non-zero status aborts the commit
    - git symbolic-ref --short HEAD: gets the branch name corresponding with HEAD
    - this script will always run when commit\_type is equal to message (-m)
    - without the –m option, the prepare-commit-msg allows the user to edit the message after its generated
  + commit message: this hook is called after the user enters a commit message, and is an appropriate place to warn developers that their message doesn’t adhere to your team’s standards
    - this hook takes one argument: the name of the file that contains the message
    - if the hook doesn’t like the message, it can alter the file in place or just simply abort the commit with a non-zero status
    - this hook shouldn’t do much outside of checking the commit message for your team’s standards
  + post-commit: called immediately after the commit-msg hook, but it can’t change the outcome of the git commit operation, so It’s primarily used for notification purposes
    - this hook takes no parameters and has no effect on commit in any way
    - git rev-parse HEAD: to get the new commit’s SHA1 hash
    - git log -1 HEAD: to get all of its information
    - it’s better to implement a continuous integration system on the server-side rather than locally
  + post-checkout: this hook is called whenever you successfully checkout a reference with git checkout, it helps clear out your working directory of generated files that could cause unnecessary confusion
    - this hook accepts 3 parameters: the ref of the previous head, the red of the new head, flag tell you if it was a branch checkout or file checkout, 1/0, respectively
    - when using python, generated .pyc files stick around after switching branches, you can delete these files every time you checkout a new branch using this hook
  + pre-rebase: called before git rebase changes anything, so it’s a good place to make sure nothing bad is about to happen
    - takes in 2 parameters: the upstream branch the series was forked from, and the branch being rebased
    - the second branch is empty when the current branch is being rebased
    - with this hook, you could completely disallow rebasing
  + server-side hooks work just like local ones, but they reside in the server side repositories
  + these can serve as a way to enforce policy by rejecting certain commits
  + pre-receive: this hook is executed every time somebody pushes commits to the repository, and it should always reside in the remote repo that is the destination
    - this hook can prevent malformed commits from entering the official codebase by rejecting them for things like bad commit message, bad changes, bad developer etc.
    - the hook takes no parameters, but instead each ref that is pushed is passed into the script to look at and inspected the changes
    - commonly this hook is used to reject changes that involves a upstream merge, preventing non-fast-forward merges, and checking that the user has the correct permissions to make the intended changes
    - if multiples refs are pushed, returning a non-zero status aborts them all, unless you want to accept/reject them on a case-by-case basis
  + update: this hook is called separately for each ref that was pushed
    - it accepts 3 arguements: the name of the ref being updated, the old object name stored in the ref, the new object name being stored in the ref
    - this hook allows you to reject some refs, while allowing others
  + post-receive: this hook gets called after successful push operations, making it a great place to perform notifications
    - emailing other developers and triggering a continuous integration system are common used for this hook
    - this hook takes in no inputs but is sent the same information as pre-receive via standard input
* Refs and the Reflog
  + hashes: a SHA-1 hash acts as the unique ID for each commit, you can find this in the git output log
    - when passing a commit to git command, you only need to specify enough characters to uniquely identify that commit
    - git rev-parse: useful for resolving a branch, tag, or another indirect reference into the corresponding commit hash, useful when writing custom scripts that accept commit references, since this is automatic instead of manual
  + refs: an indirect way of referring to commits, a user-friendly alias for a hash, and git’s internal mechanism of representing branches and tags
    - the heads directory contains all of the local branches in your repository and inside each branch file, you will find a commit hash that corresponds to the tip of the branch
    - therefore to change the location of branches, all git has to do is change the contents of the branch file
    - creating a new branch is simply writing a commit hash to a new file
    - the tags directory works the same way, but contains tags instead of branches
    - the remote directory contains all of the remote repositories as well as all of the remote branches that have been fetched into your repository
    - git show <branch>: allows you to specify refs by using short names for branches
  + packed refs: when the repository gets large, git periodically performs garbage collection to remove unnecessary objects and compress refs into a single file called packed-refs for more efficient performance
    - git gc: allows you to force garbage collection
  + special refs: refs created and updated by git
    - HEAD: the current checked out branch/commit
    - FETCH\_HEAD: the most recent fetched branch from a remote repo
    - ORIG\_HEAD: a backup reference to head before drastic changes to it
    - MERGE\_HEAD: the commits you are merging into the current branch with git merge
    - CHERRY\_PICK\_HEAD: the commit you are cherry-picking
    - these file contains different content depending on the type and state of your repository
    - symbolic ref: a commit hash or a reference to another ref instead of a commit hash
  + refspecs: a refspec maps a branch in a local repository to a branch in a remote repository, making it possible to manage remote branches using local git commands and to configure some advanced git push/fetch behavior
    - a refspec is specified as [+]<src>:<dst>, where <src> is the source branch in the local repository, <dst> is the destination branch in the remote repository, and the optional [+] forces the remote repository to perform a non-fast forward update
    - refspecs can be used with git push to give a different name to the remote branch
    - you can delete dead feature branches by pushing a refspec that has an empty <src> parameter
    - you can also use the --delete flag instead of a refspec
    - you can also use refspecs to alter the behavior of git fetch and git push, so that it only fetches the master branch or always pushes to a specified branch
  + relative refs: the ~ character lets you refer to commits relative to another commit, allowing you to reach parent commits
    - when dealing with merge commits, things get more complicated because you have 2 parents
    - the ~ character will always follow the first parent, while the ^ character allows you to specify which parent you want to follow
    - to get to the a grandparent of the second parent you need to use ^2^1 or ^2~1
  + reflog: the reflog records almost every change you make in your repository, regardless of whether you committed a snapshot or not
    - git reflog: chronological history of everything you’ve done in your repo
    - HEAD{<n>}: lets you reference commits stored in the reflog
    - the reflog is useful to revert to a state that would otherwise by lost