# Most Common/Useful Commands:

<https://github.com/git-tips/tips>

<https://jwiegley.github.io/git-from-the-bottom-up/>

* Help:
* *git help* [-a|--all] [-g|--guide]
* [-i|--info|-m|--man|-w|--web] [COMMAND|GUIDE]
* Create a new branch and move to it:

$ git checkout –b <new branch name>

* Locations relative to current HEAD (the symbolic name for the currently checked out commit):
  1. The direct parent of the current branch/HEAD: master^ / HEAD^
  2. X parent of the current branch/HEAD: master~X / HEAD~X
* See remote repository:

>> git remote -v

* Discard all changes since last commit and reverts the repo to the state of the HEAD revision, which is the last committed version:   
  >> git reset --hard
* Create patches for all commits:  
  >> git format-patch <name of branch to compare to> -o <name of directory to put the patches from>
* Apply patch:  
  >> git am <name of patch file>
* See all the branches:  
  >> git branch -a -v
* Add (files to commit) – will add all changed files to commit:  
  >> git add .
* Commit:  
  >> git commit -m “commit message”
* Add to previous commit (instead of creating a new one):  
  >> git commit --amend
* Squash commits:  
  <https://www.internalpointers.com/post/squash-commits-into-one-git>
  1. Send: git rebase --interactive [commit-hash]

where [commit-hash] is the has of the commit just before the first one you want to re-write from.

* 1. In the opened squashing file, choose the commits you want to squash. You should leave the 1st one as pick and change the following commits to 's' to mark 'squash'
  2. In the automatically opened commit file, write the commit message for the united commit.
* Revert the last commit (without undoing the changes, just get all the files back to the staging area):  
  git reset --soft HEAD^
* Undo a ‘git add’ for a specific file:  
  git reset {filename}
  1. Or for all files with:  
     get reset
* Revert changes for a single file:  
  >> git checkout – {file name}
* Pretty log:  
  will print out the last 10 commits with the following format:  
  {short hash} {commit date} {author} {commit subject line}  
  git log -10 --pretty=format:"%h %cd %an %s"  
  For additional formats: <https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History>
* See all the files that were included in the last commit:  
  git log --name-status HEAD^..HEAD
  1. Note: HEAD^..HEAD recognise the last commit – you can use it to do other things like get the status of the last commit (git status…) , git diff… etc.

# GIT Tutorial:

Great tutorial: <http://learngitbranching.js.org/>

## git init – Create a bare repository

Create a new git repository. Can be used to create a completely new repository or to transfer an existing non-source control directory into git.

### Local repository – to connect to a remote repository and pull/push from :

1. Create the new repository:

>> mkdir <directory for new repository>

>> cd <directory for new repository>

>> git init

1. Connect to the remote repository – see *Connect*

Notes:

* To remove a repository you previously creates, delete the .git directory:

>> rm -rf .git

### Main repository – that you can clone and commit into:

Note: if you create a regular repository, you’ll have problems pushing into it because git will complain that it’s checked out. In order to avoid this problem, it’s usually a good idea to create a **bare repository**

In order to create a bare repository (never checked out):

1. Create a directory where you want your repository to be:  
    *mkdir test\_repo.git // It's conventional to give bare repositories the extension .git.*
2. *Go into the* directory *and turn it into a GIT bare repository:  
    cd test\_repo.git  
    git --bare init*
3. Go to where you want to work on and clone the bare repository (normal clone).
4. Copy/Create files in the cloned repository
5. commit and push to bare repository.

Bare repositories (by definition) don't have a working tree attached to them, so you can't easily add files to them as you would in a normal non-bare repository (e.g. with git add <file> and a subsequent git commit.)

## git clone –

Creates a development copy of the repository. Similar to SVN checkout, except the “working copy” is a full-fledged Git repository—it has its own history, manages its own files, and is a completely isolated environment from the original repository.

As a convenience, cloning automatically creates a remote connection called **origin** pointing back to the original repository. This makes it very easy to interact with a central repository.

Like [git init](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init), cloning is generally a one-time operation—once a developer has obtained a working copy, all version control operations and collaborations are managed through their local repository.

Create a ‘repo-name’ directory in the current location and clone the ‘repo-name’ into it.

>> git clone <link to the repo/repo\_name>

Create a ‘my-name’ directory in the current location and clone the ‘repo-name’ into it.

>> git clone <link to repo/repo\_name> <my-name

### Repo-To-Repo Collaboration

It’s important to understand that Git’s idea of a “working copy” is very different from the working copy you get by checking out code from an SVN repository. Unlike SVN, Git makes no distinction between the working copy and the central repository—they are all full-fledged Git repositories.

This makes collaborating with Git fundamentally different than with SVN. Whereas SVN depends on the relationship between the central repository and the working copy, Git’s collaboration model is based on repository-to-repository interaction. Instead of checking a working copy into SVN’s central repository, you [push](https://www.atlassian.com/git/tutorials/syncing/git-push) or [pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) commits from one repository to another.

## Staging

### The Staging Area

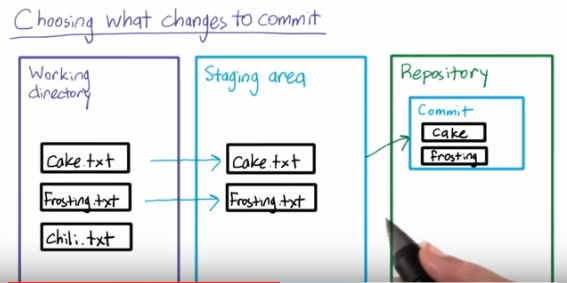
The staging area is one of Git's more unique features, and it can take some time to wrap your head around it if you’re coming from an SVN (or even a Mercurial) background. It helps to think of it as a buffer between the working directory and the project history.

Instead of committing all of the changes you've made since the last commit, the stage lets you group related changes into highly focused snapshots before actually committing it to the project history.

This means you can make all sorts of edits to unrelated files, then

**go back and split them up into logical commits by adding related changes to the stage and commit them piece-by-piece**.

As in any revision control system, it’s important to create atomic commits so that it’s easy to track down bugs and revert changes with minimal impact on the rest of the project.



To stage a file to be committed:

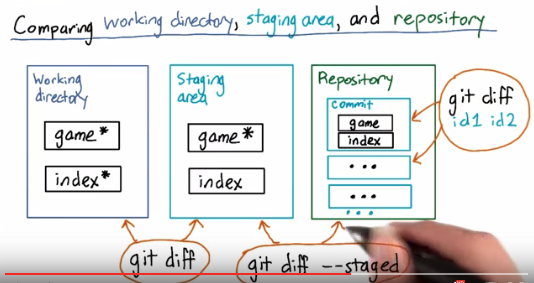
>> git add <file>

To remove a file from the staging area (doesn’t delete the file just unstage it!):

>> git reset <file>

## Diff

To compare between two versions of the repository:



>> git diff <id1> <id2> - will show the differences between the two commits in the repository.

>> git diff – without any argument will show the differences between the working directory and the staging area

>> git diff –staged – will should the differences between the staging area and the repository.

If you want to see the diff of a commit but you don’t know who their parent it, run:

>> git show commit\_id

## Commit

The git commit command commits the staged snapshot to the project history. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicity ask it to.

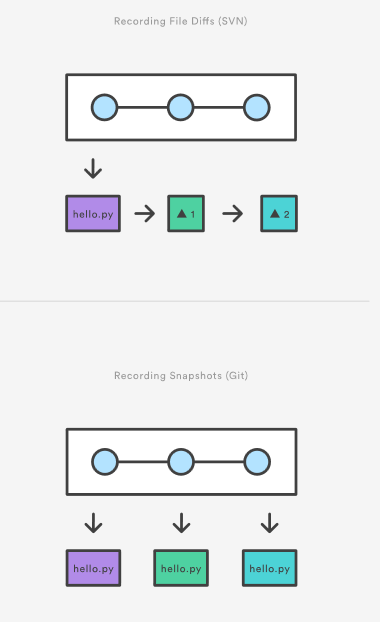
While they share the same name, this command is nothing like svn commit.

**Snapshots are committed to the local repository, and this requires absolutely no interaction with other Git repositories.**

This changes the basic development model for Git users. Instead of making a change and committing it directly to the central repo, Git developers have the opportunity to accumulate commits in their local repo. This has many advantages over SVN-style collaboration: it makes it easier to split up a feature into atomic commits, keep related commits grouped together, and clean up local history before publishing it to the central repository. It also lets developers work in an isolated environment, deferring integration until they’re at a convenient break point.

### Snapshots, Not Differences

Aside from the practical distinctions between SVN and Git, their underlying implementation also follow entirely divergent design philosophies. Whereas SVN tracks *differences* of a file, Git’s version control model is based on *snapshots*. For example, an SVN commit consists of a diff compared to the original file added to the repository. **Git, on the other hand, records the *entire contents* of each file in every commit.**



### Partial Commits

When you have changes that related to different things and you want to split them between a few commits, you can commit only parts of the files.

You can do git add --patch filename.x (or -p for short), and git will begin breaking down your file in what it thinks are sensible "hunks" (portions of the file). You will then be prompted with this question:

Stage this hunk [y,n,q,a,d,/,j,J,g,s,e,?]?

And here the meaning of each option:

* y stage this hunk for the next commit
* n do not stage this hunk the next commit
* q quit; do not stage this hunk or any of the remaining ones
* a stage this hunk and all later hunks in the file
* d do not stage this hunk or any of the later hunks in the file
* g select a hunk to go to
* / search for a hunk matching the given regex
* j leave this hunk undecided, see next undecided hunk
* J leave this hunk undecided, see next hunk
* k leave this hunk undecided, see previous undecided hunk
* K leave this hunk undecided, see previous hunk
* s split the current hunk into smaller hunks
* e manually edit the current hunk
* ? print help

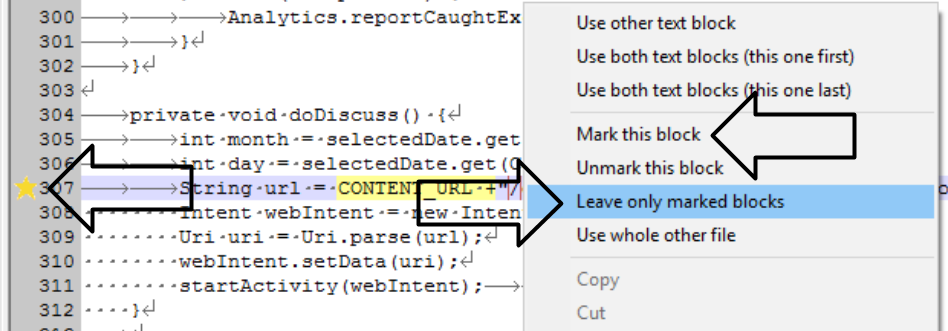
If the file is not in the repository yet, do first git add -N filename.x. Afterwards you can go on with git add -p filename.x.

You can use than: git diff --staged afterwards to check that you staged the correct ones git reset -p to unstage incorrect hunks git commit -v to view your commit while you edit the commit message.

Note this is a far different than the git format-patch command, which is entirely different.

In Tortoise GIT:

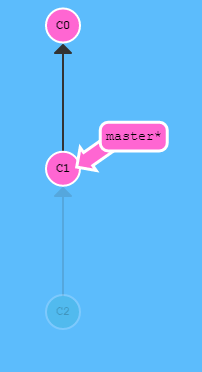
* **Right click** a file you want to partially commit
* Click **Restore after commit**
* **Double click** the file to edit in **TortoiseGitMerge**
* **Right click** -> **Mark this block** for each change you want to commit now
* **Right click** -> **Leave only marked blocks** to revert the other changes
* **Save** and **Close TortoiseGitMerge**
* **Commit**
* The changes you just reverted are now restored in your working tree.

[](http://i.stack.imgur.com/8QUxL.png)

## Undo Changes

### Get Reset – Delete Changes Forever (Local Branch Only)

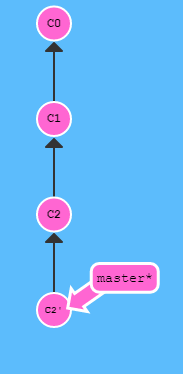
$ git reset HEAD~1



Reverts changes by moving a branch reference backwards in time to an older commit. In this sense you can think of it as "rewriting history;" git reset will move a branch backwards as if the commit had never been made in the first place.

### Get Revert

$ git revert HEAD



While reseting works great for local branches on your own machine, its method of "rewriting history" doesn't work for remote branches that others are using.

In order to reverse changes and *share* those reversed changes with others, we need to use git revert

The git revert command undoes a committed snapshot. But, instead of removing the commit from the project history, it figures out how to undo the changes introduced by the commit and appends a *new* commit with the resulting content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.

### Reverting vs. Resetting

It's important to understand that git revert undoes a single commit—it does not “revert” back to the previous state of a project by removing all subsequent commits. In Git, this is actually called a [reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset), not a revert.

Reverting has two important advantages over resetting. First, it doesn’t change the project history, which makes it a “safe” operation for commits that have already been published to a shared repository. For details about why altering shared history is dangerous, please see the [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) page.

Second, git revert is able to target an individual commit at an arbitrary point in the history, whereas git reset can only work backwards from the current commit. For example, if you wanted to undo an old commit with git reset, you would have to remove all of the commits that occurred after the target commit, remove it, then re-commit all of the subsequent commits. Needless to say, this is not an elegant undo solution.

## git reset

If git revert is a “safe” way to undo changes, you can think of git reset as the *dangerous* method. When you undo with git reset(and the commits are no longer referenced by any ref or the reflog), there is no way to retrieve the original copy—it is a *permanent* undo. Care must be taken when using this tool, as it’s one of the only Git commands that has the potential to lose your work!

## Push and Pull

To push your changes into the server so that all the users will be able to access it: right click -> Push

To update your local copy with all the changes from the server: right click -> Pull. In order to

## Branches

See all the branches (both local and remote):

git branch –a -v

create a new branch:

git branch <name of new branch>

delete branch:  
git branch -d branch\_name

**Branch:**

1. Create a new local branch: right click->Create Branch. Choose switch to new branch if needed.
2. Push new branch to the server (when required).
3. Pushing additional changes: make sure you choose the current remote branch when pushing.

If another user wants to work on this branch, they will do: right click -> fetch

## Merging

1. Switch back to the master branch (before the branch) right click -> Checkout\Switch
2. Right click -> Merge …   
   choose the branch we want to merge in the From section

* Choosing ‘No fast forward’ will ensure we will not lose information about the historical existence of the branch. Making it easier to debug and revert changes if needed later on.

1. Push changes into the server (master)

## Heads

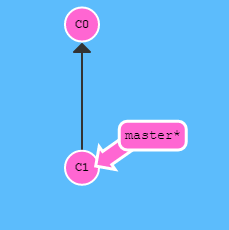
HEAD is the symbolic name for the currently checked out commit -- it's essentially what commit you're working on top of.

Normally HEAD points to the branch that you are working on.

### Detaching Head

Detaching HEAD just means attaching it to a commit instead of a branch. This is what it looks like beforehand:

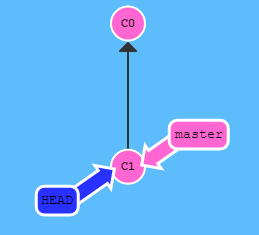
HEAD -> master -> C1



$ git checkout C1  
the command is: $ git checkout <commit’s hash>

And now it's

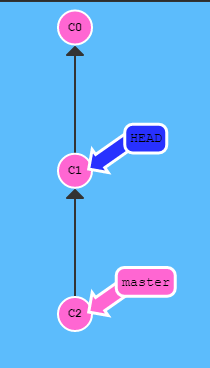
HEAD -> C1



### Detach to Branch/Head Ancestors

* Set HEAD to the current branch’s parent:

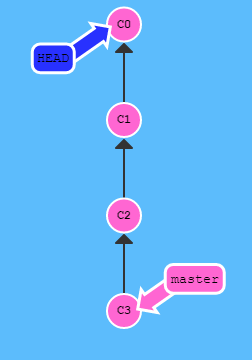
$ git checkout master^



* Set HEAD to the current HEAD’s parent:

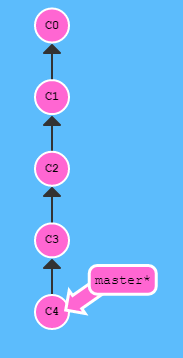
$ git checkout HEAD^

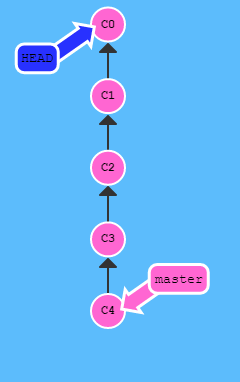
We can use this to travel back in time:



* Set HEAD to 4 parents before the current HEAD:

$ git checkout HEAD~4





## Branch Forcing

Advanced Topic. Be Carefull

You can directly reassign a branch to a commit using detaching heads and branch forcing:

$ git branch –f <branch-name> HEAD~3

Will force <branch-name> branch to point to three parents before HEAD.

You can use any of the methods (commit checksum, ^,~,branch name) for detaching the HEAD to define the new branch position.

## Delete Branches

After we’ve merged the branches into the master, we’d want to delete them. This is how:

1. Right click -> Checkout\Switch. Press the ‘…’ near the branch, choose the branch(es) local and remote , right click -> Delete Branch.

## Suggested Schema for Development:

### Feature Branch Workflow

##### How it works

* All development for a new feature is performed in a dedicated feature branch.
* This allows multiple developers the ability to iterate on a feature without modifying master.
  + \_Hopefully\_ this means that master never gets incomplete code, making it advantageous for continuous integration/deployed environments.
* This feature encapsulation allows teams to utilize pull requests.
  + This allows teams to discuss and review code before it’s merged into master.
* Some workflows build on top of this workflow, such as Gitflow and Forking Workflow.

##### Example

* Start from master (i.e. git checkout master).
* Create a new branch off of master (i.e. git checkout -b MyNewFeature).
* Similar to before, make, stage, and commit changes (i.e. git add, git commit).
* Push your new branch to the remote (or centralized) repo (i.e. git push origin MyNewFeature).
* Use the tooling of your repo management system to create a PR (i.e. GitHub, GitLab, VSTS, Bitbucket, etc.)
  + Note that if there are conflicts, those will need to be corrected locally (i.e. git pull –rebase again) and the feature branch re-pushed.
* Prior to the pull request, if another developer wanted to contribute to the feature branch, they could pull it locally to contribute to it (i.e. git pull MyNewFeature) assuming the feature branch has already been pushed to the centralized repo.

##### Pros

* Promotes code review and team collaboration.
* Keeps master stable.

##### Cons

* Long lived branches have a higher risk of merge conflicts, best to keep your features small.

##### When should the seasoned Git guru use it?

* Always? /8)
  + This workflow is likely the building block to another workflow you need.
* Definitely best used for large teams and/or projects.

### Gitflow

##### How it works

* Builds upon the Feature Branch workflow.
* However instead of just a master branch, you create additional branches with very specific purposes.
  + As well as specific times when those branches should interact.
* The master branch represents the officially released versions.
  + Create version tags off of master.
* A develop branch is used to iterate on the next version.
  + All new feature branches are based on develop instead of master (unlike the Feature Branch workflow).
* When it’s time to release, a new release branch is based off of develop.
  + Fixes to the release are made directly to the release branch.
* When the release has been tested and ready for production (i.e. release to manufacture, etc). the release branch is merged into both master and develop.
  + This is when you tag master with the version number.
* Hot fixes are made in separate branches based off of master.
  + These are the only branches that are based off of master.
  + Once the hot fix is complete, the hot fix branch is merged into both master and develop
  + Tag the version in master.

##### Pros

* Ideal for projects with a scheduled release cycle.
* Great for projects where you only need to support one version, for example, your company’s website.
* All of the benefits of the Feature Branch workflow, i.e. pull requests, isolated feature development, collaboration.
* Tooling exists to help streamline the process.
* Master only has stable code.
* Allows one team to polish the release while another(s) can work on features for the next release.
* Hot fix branches allow teams to address issues without disrupting other teams working to polish a build or develop new features for the next version.
  + Atlassian refers to this benefit as a “dedicated channel for hot fixes to production.“
* No commits left behind.

##### Cons

* Doesn’t work well with projects that need to support multiple versions concurrently, for example, supporting code for all currently supported versions of Windows.
  + Welcome to Merge Hell. Population: You.

##### When should the seasoned Git guru use it?

* Works great for projects that tend to roll forward.



Release branches support preparation of a new production release. They allow for last-minute dotting of i’s and crossing t’s. Furthermore, they allow for minor bug fixes and preparing meta-data for a release (version number, build dates, etc.). By doing all of this work on a release branch, the develop branch is cleared to receive features for the next big release.

The key moment to branch off a new release branch from develop is when develop (almost) reflects the desired state of the new release. At least all features that are targeted for the release-to-be-built must be merged in to develop at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

This new branch may exist there for a while, until the release may be rolled out definitely. During that time, bug fixes may be applied in this branch (rather than on the develop branch). Adding large new features here is strictly prohibited. They must be merged into develop, and therefore, wait for the next big release.

### Microsoft’s Recommendations

##### How it works

* Similar to Gitflow, [Microsoft’s branching guidance](https://docs.microsoft.com/en-us/azure/devops/repos/git/git-branching-guidance?view=vsts) will use release branches (plural!).
* The biggest difference with this strategy compared to other workflows is the lack of merging (yay!).
* Similar to the Feature Branching workflow, you’ll create feature branches off of master.
  + These feature branches are the only branches that get merged.
* When it’s time for a release, you create a new release branch for that version (i.e. the version will be part of the branch name).
* Any fixes to a release should be cherry picked to all of the other relevant branches that need the fix
* No need for tagging versions.

##### Example

* The new command here is [git cherry-pick](https://git-scm.com/docs/git-cherry-pick).
  + Michael typically includes [-x](https://git-scm.com/docs/git-cherry-pick) when using the command line.

##### Pros

* No more merge hell.
* Total control over which commits get merged and where they are merged to.
* No “I forgot to push the latest tag” issues.
* Works well for large projects/teams.
* Perfect for software with multiple versions that need to be supported concurrently.
* When there is a merge conflict, \_you\_ know the details (i.e. \_your\_ code is involved).

##### Cons

* Very easy to forget to cherry pick a commit to every branch it needs to go to (or maybe you didn’t even know all of the branches it needs to go to).

##### When should the seasoned Git guru use it?

* Great when you need to support multiple concurrent releases

## Finishing a release branch

When the state of the release branch is ready to become a real release, some actions need to be carried out. First, the release branch is merged into master (since every commit on master is a new release *by definition*, remember). Next, that commit on master must be tagged for easy future reference to this historical version. Finally, the changes made on the release branch need to be merged back into develop, so that future releases also contain these bug fixes.

The release is now done, and tagged for future reference.

## Submodules (Externals)

A submodule allows you to keep another Git repository in a subdirectory of your repository. The other repository has its own history, which does not interfere with the history of the current repository. This can be used to have external dependencies such as third party libraries for example. Similar to SVN externals.

When cloning or pulling a repository containing submodules however, these will not be checked out by default; the init and update subcommands will maintain submodules checked out and at appropriate revision in your working tree.

With submodules:

* The history of the two projects still stays completely independent and
* You cannot modify the contents of the submodule from within the main project

If you want to merge the project histories and want to treat the aggregated whole as a single project from then on, you may want to add a remote for the other project and use the subtree merge strategy.

To add a new submodule to your project:

>> git submodule add <repository> <directory>

## SubTrees

If you want to split a subdirectory of an existing repository and move it into it’s own repository, you need to follow the steps below using GIT subtrees:

This is such a common and useful practice that the overlords of git made it really easy, but you have to have a newer version of git (>= 1.7.11 May 2012). See the **appendix** for how to install the latest git. Also, there's a **real-world example** in the **walkthrough** below.

1. Prepare the old repo

cd <big-repo>

git subtree split -P <name-of-folder> -b <name-of-new-branch>

**Note:** <name-of-folder> must NOT contain leading or trailing characters. For instance, the folder named subproject MUST be passed as subproject, NOT ./subproject/

**Note for windows users:** when your folder depth is > 1, <name-of-folder> must have \*nix style folder separator (/). For instance, the folder named path1\path2\subproject MUST be passed as path1/path2/subproject

1. Create the new repo

>> mkdir <new-repo>

>> cd <new-repo>

>> git init

>> git pull </path/to/big-repo> <name-of-new-branch>

1. Link the new repo to Github or wherever:

>> git remote add origin <git@github.com:my-user/new-repo.git>

>> git push origin –u master

1. Cleanup, if desired:

>> cd <big-repo>

>> git rm -rf <name-of-folder>

**Note**: This leaves all the historical references in the repository.See the **Appendix** below if you're actually concerned about having committed a password or you need to decreasing the file size of your .git folder.

# Useful Commands

Additional useful commands:  
<https://github.com/tj/git-extras>

## Remote Repository

### Connect

To connect your local repository to a remote repository (e.g. GitHub):

>> git remote add <name for the remote repository e.g. origin> <remote repository url>

When you clone a repository using “git clone” command, GIT does this automatically and created a remote connection called **origin** which points back to the cloned repository.

To see all the remote repositories:

>> git remote -v

To see more details details regarding the remote repository:

>> git remote show <name of remote repository e.g. origin>

To see which remote repository is the default repository for each branch:

>> branch -vv

To set which remote repository is the default remote repository for pull and push commands:

>> git push -u <remote\_name> <local\_branch\_name>:<corresponding\_remote\_branch\_name>

## Local Repository

If you have two repositories which are clones of the same remote repository and you want to synchronize them,  
You can treat the second clone the same way you treat a remote repository on another system. You can perform all of the same operations, e.g.

~/repo1 $ git remote add repo2 <repo2 path>

~/repo1 $ git fetch repo2

~/repo1 $ git merge repo2/foo

## Log

>> git log [optional: -n <num of commits to show> ]

To show a graphic representation of the branches and commits:

>> git log --graph --oneline <branch 1> <branch 2>

* To show the logs of only part of a file (for example, function in a file):  
  git log -L :{function}:{file}

Additional options:  
git log -L <start>,<end>:<file>

git log -L :<funcname>:<file>

git log -S<string>

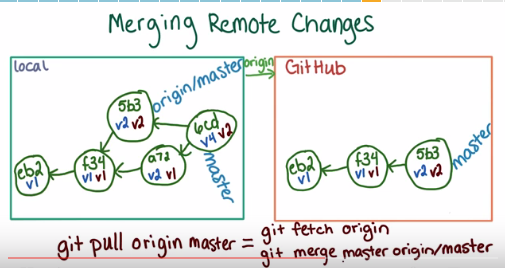
git log -G<regex>

## Fetch

Imports commits from a remote repository into your local repo. The resulting commits are stored as remote branches instead of the normal local branches that we’ve been working with. This gives you a chance to review changes before integrating them into your copy of the project (like SVN update):

Fetch all branches from remote:  
>> git fetch <remote>

Fetch specific branch from remote:  
>> git fetch <remote> <branch>



>> git branch –a will show both ‘master’ and ‘origin/master’ branches.

## Pull

To merge upstream changes into your local repository.  
 >> git pull <remote-repository> <branch on repository>

-rebase will make sure that all of your changes are located together at the end and not intermitted in the changes from the server (according to the date and time). This is very recommended since it makes reviewing and searching for the changes much easier.

## Push

>> git push <remote-repository> <branch> --follow-tags

--follow-tags will also push all the annotated tags that belongs to this branch

## show all recent activity

>> git status – show your current state (if there are changes to commit etc)

>> git reflog

>> git log –g

>> git log --walk-reflogs

## Branches:

**Branch early, branch often**

Branches in Git are incredibly lightweight. They are simply pointers to a specific commit – nothing more.

Create a new branch:  
>> git branch <new branch>

Show all branches (active one marked with \*):  
>> git branch

Show all remote branches (active one marked with \*):  
>> git branch -r

Delete a branch (Safe command – will only work if already merged. Otherwise will return an error):  
>> git branch –d <branch name>

## Checkout:

Switch to branch: Update working directory to point to and be updated with the specified branch:  
>> git checkout <existing branch>

**Create a new branch and move to it:  
>> git checkout –b <new branch>**

## Revert Local Changes

If you want to get rid of all local changes and go back to the last clean checkout, send:

>> git checkout .

If you want to get rid of local changes that you haven’t staged for commit yet:

git checkout -- .

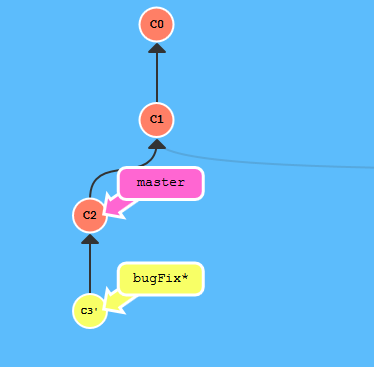
## Merge

1. Change directory to the branch into which you want to merge:  
   >> git checkout <main branch>
2. Merge the branch into the main branch:  
   >> git merge <branch>  
   or  
   >> git merge –no-ff <branch> - create a merge commit – useful for documentation.git

## Rebasing

The second way of combining work between branches is *rebasing.* Rebasing essentially takes a set of commits, "copies" them, and plops them down somewhere else.

While this sounds confusing, the advantage of rebasing is that it can be used to make a nice linear sequence of commits. The commit log / history of the repository will be a lot cleaner if only rebasing is allowed.



$ git rebase <the name of the branch into which we rebase>

In the above picture:

$ git checkout bugFix

$ git rebase master

## Tags

See all tags in the repository:

>> git tag [-l <pattern>]

#### Lightweight Tag

A lightweight tag is very much like a branch that doesn’t change – it’s just a pointer to a specific commit.  
Usually used for temporary tags. Not recommended for permanent tags.

#### Annotated Tag

Annotated tags, however, are stored as full objects in the Git database. They’re checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It’s generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don’t want to keep the other information, lightweight tags are available too.

* In order to tag a specific version:
* Annotated tag (recommended) – contains full version objects that can be verified etc:

>> git tag -a <tag label> -m <'tag message'>

* To see the details of the tag:

>> git show <tag label>

* Push the tag to the server:

>> git push origin <tag label>

* Tagging after the fact:   
  In order to tag a previous commit:

>> git tag -a <tag label> -m <'tag message'> <commit checksum 4 first bytes >

* Lightweight Tags:  
  Don’t include the ‘-a’ and ‘-m <’tag message’>’:

>> git tag <tag label>

* In order to checkout a specific tag:

>> git checkout -b <branch name> <tag label>  
 example: git checkout -b version2 v2.0.0

## Stashing

Often, when you’ve been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don’t want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the git stash command.

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time.

>> git stash - now all your local changes are saved in the git stack and your working directory is  
 clean (identical to the server before your local changes)

>> git stash save "my\_stash" – will name your stash for easier recognition.

>> git stash list - returns a list of all your stored stashes

>> git stash apply - will reapply the last saved stash

>> git stash apply <stash@{num}> - will apply the stash number {num} (as appears in the list command)

>> git stash drop <stash@{num}> - will delete the specified stash

>> git stash pop – apply the top stash and remove it from the stack immediately

To stash your current repository and move it to a new branch:

>> git stash  
>> git stash branch <name of new branch>

This will:

1. Creates a branch from your current working directory (cleaned after the stash command)
2. Move into that branch
3. Merge your last stash into that branch.  
    NOTE: you should send ‘git stash’ before sending this command to create the stash to be moved into the new branch
4. If the merge was successful, remove the stash from your stack.  
   \*\* this is an excellent solution if you made some changes and then decided you want to branch them.

# Git Configuration

## Git Attributes

. gitattributes file in the root directory of a repository can use to modify the default way in which git handles this repository.

See <http://schacon.github.io/git/gitattributes.html>

# Style

## Udacity Git Commit Message Style Guide

### Introduction

This style guide acts as the official guide to follow in your projects. Udacity evaluators will use this guide to grade your projects. There are many opinions on the "ideal" style in the world of development. Therefore, in order to reduce the confusion on what style students should follow during the course of their projects, we urge all students to refer to this style guide for their projects.

### Commit Messages

#### Message Structure

A commit messages consists of three distinct parts separated by a blank line: the title, an optional body and an optional footer. The layout looks like this:

type: subject

body

footer

The title consists of the type of the message and subject.

#### The Type

The type is contained within the title and can be one of these types:

1. **feat:** a new feature
2. **fix:** a bug fix
3. **docs:** changes to documentation
4. **style:** formatting, missing semi colons, etc; no code change
5. **refactor:** refactoring production code
6. **test:** adding tests, refactoring test; no production code change
7. **chore:** updating build tasks, package manager configs, etc; no production code change

#### The Subject

Subjects should be no greater than 50 characters, should begin with a capital letter and do not end with a period.

Use an imperative tone to describe what a commit does, rather than what it did. For example, use **change**; not changed or changes.

#### The Body

Not all commits are complex enough to warrant a body, therefore it is optional and only used when a commit requires a bit of explanation and context. Use the body to explain the **what** and **why** of a commit, not the how.

When writing a body, the blank line between the title and the body is required and you should limit the length of each line to no more than 72 characters.

#### The Footer

The footer is optional and is used to reference issue tracker IDs.

#### Example Commit Message

feat: Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72

characters or so. In some contexts, the first line is treated as the

subject of the commit and the rest of the text as the body. The

blank line separating the summary from the body is critical (unless

you omit the body entirely); various tools like `log`, `shortlog`

and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you

are making this change as opposed to how (the code explains that).

Are there side effects or other unintuitive consequenses of this

change? Here's the place to explain them.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded

by a single space, with blank lines in between, but conventions

vary here

If you use an issue tracker, put references to them at the bottom,

like this:

Resolves: #123

See also: #456, #789

# Troubleshooting

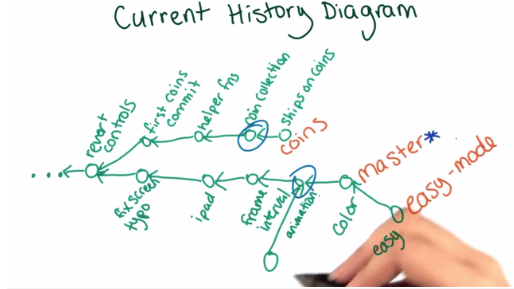
**I lost my commit:**

GIT garbage collect dead-end commits after two weeks. To try to recover your commit:

1. git log --walk-reflogs – see if you can locate the commit in the list. If yes, send:
   1. git merge <commit number as appears in the previous command>

##### Detached HEAD

When you do a checkout from a commit and not a branch, git will warn you that you have created a ‘detached HEAD’. This means that your checkout and subsequent commits will not be reachable from any branch in the repository:



You can experiment and then revert your checkout or,

You can attach your checkout by creating a branch on it.

##### Password Caching

To tell GitHub to cache and remember your credentials on your computer:

git config --global credential.helper wincred

# GitHub

## Fork

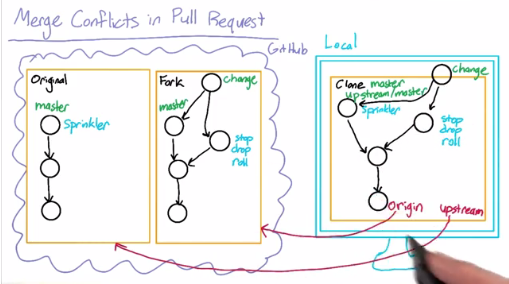
Copy a repository on GitHub to a different repository on GitHub and keeps a connection between the forked repository and its ‘source’.

## Collaboration on GitHub

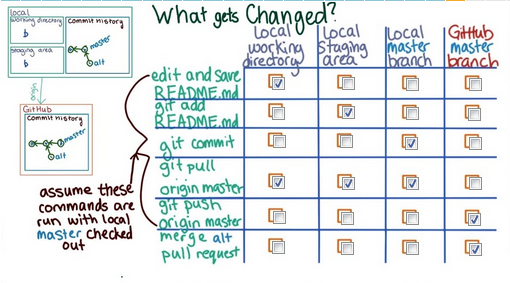
The main steps in collaborating of GitHub:

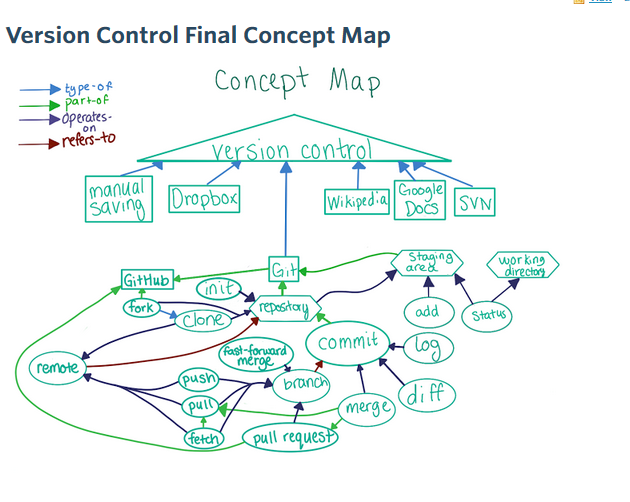
1. Create/Fork a repository (in GitHub)
2. Add collaborators in GitHub->repository->Settings. From here on, I assume that you forked a public repository.
3. Clone it to your computer. This will automatically create the remotes/origin connection to your forked repository
   1. Connect remotes/upstream to the original public repository to enable publishing your changes:  
      >> git remote add <upstream> <url of original repository>
4. When you develop a new feature:
   1. Create and checkout a new branch
   2. Commit and push the new branch. When you’re ready to merge your changes into the master:
   3. Pull the latest version from remotes/upstream/master to your master – to keep your master up to date with the public repository.
   * Don’t forget to push your updated master to your forked repository (origin) so it will also be up to date.
   1. Merge your master into you branch to verify before pushing back to your fork.  
      \*\* Also do this synchronization with the upstream and origin master occasionally if you’re working on your branch for a long time.
   2. In GitHub create a pull request to the upstream (original public repository).  
      This will automatically send an email to the collaborators on the repository with the pull request.
   3. The reviewers can review your changes and leave comments both in commit and lines level for you to see.
   4. If you update and commit as a result of the review, your pull request will get updated automatically.
   5. When the branch is ready to be pulled into the repository, the reviewer can choose the button ‘merge pull request’

**Note: When working with a public repository (using fork):** You could make your changes directly to the master branch in your fork, but when contributing to a public repository, it’s standard practice to make the changes in a non-master branch within the fork. This way, you can easily keep your master branch up-to-date with master of the original repository, and merge changes from master into your branch when you are ready.



1. After the branch is merged into the master, you should delete the branch to keep the repository tidy.





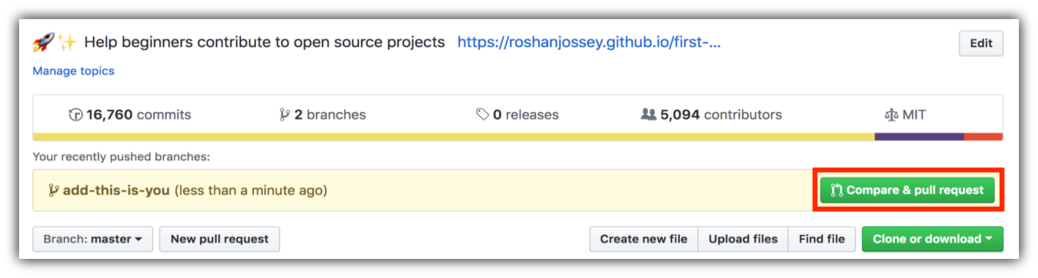
## Contribute to Open Projects on GitHub

See <https://www.firsttimersonly.com/> , <https://yourfirstpr.github.io/> etc.

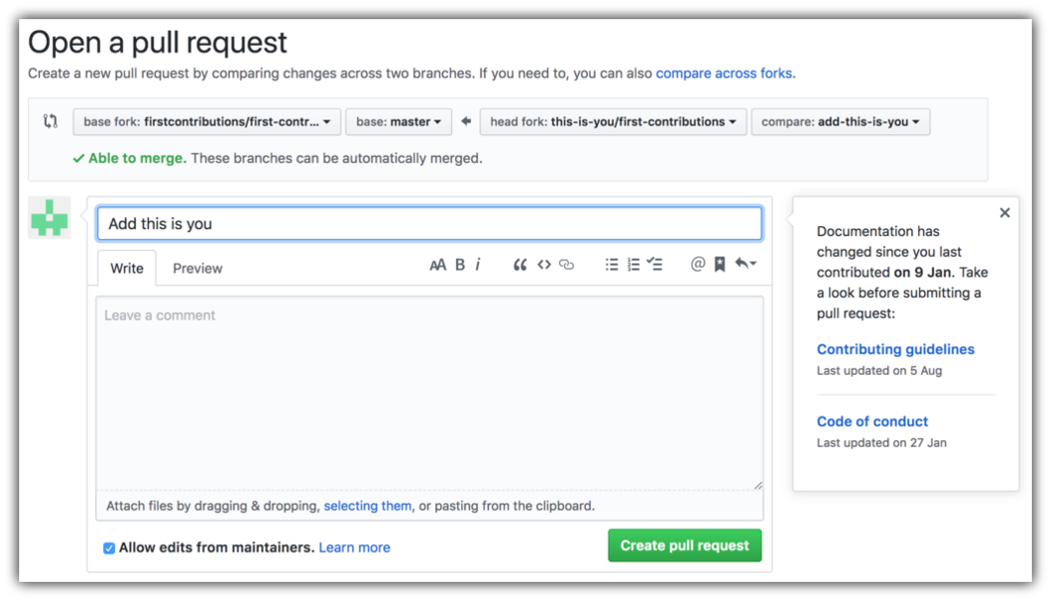
1. Read the documentation on the project:
   1. What is their license
   2. What is their contribution procedure?
   3. Do they have a code of conduct?
2. Fork the open source project.
3. Clone it to your computer
4. Always create a new branch for your changes!
5. Find what you want to fix and fix it on your fork.
6. Add tests if relevant and update the documentation.
7. Commit your changes to your fork
8. Open a ticket explaining what you want to change. This can be a documentation or code change. This will get the conversation going with the developers and (maybe) give you the permission to pull request your change.
9. Create a pull request:  
   <https://help.github.com/articles/creating-a-pull-request/>

<https://help.github.com/articles/about-pull-requests/>

1. If you go to your repository on GitHub, you'll see a Compare & pull request button. Click on that button.

[](https://github.com/Sarah-A/first-contributions/blob/master/assets/compare-and-pull.png)

1. Now submit the pull request:

[](https://github.com/Sarah-A/first-contributions/blob/master/assets/submit-pull-request.png)



# Versions

Major.Minor.Fix e.g. 2.0.5

Major – a major breaking change –SW that depends on the previous major version might break because of this change. Example: Python 2 and Python 3.

Minor – new features added but the old features still work in the same way. The old code will still work but you might not take advantage of the new available functionality. Example: Python 2.6 and 2.7.

Fix – bug fixes in existing functionality. The old code should still work as before minus fixed bugs. Example: Python 2.6.1 and Python 2.6.2