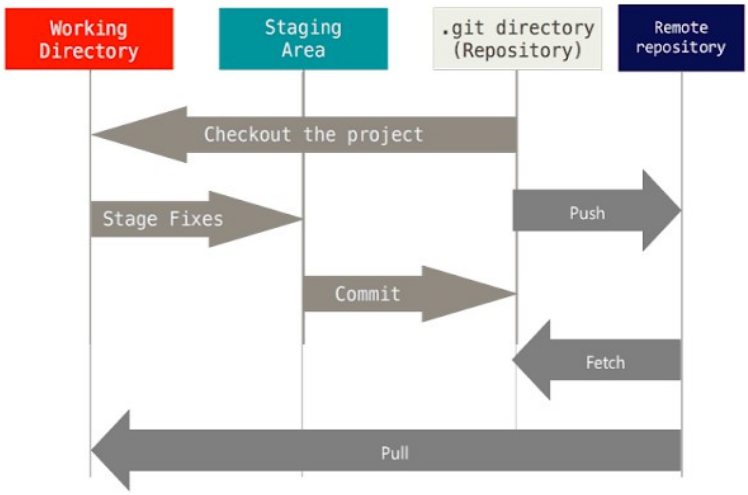
Git Guide

Git is a CLI VC program where files in a directory are tracked for changes. Git allows the user to revert files to previous versions, compare versions, etc., and stores a version history graph. There are 5 virtual stages in which a file's changes can be. Firstly, not shown in diagram (right), a file can be untracked, or even ignored (if the file is listed in the .gitignore file); the changes are in the wd when they are not staged for commitment; they are in the staging area when they are staged for commitment; they are in the (local) repository (the .git directory) once they are committed; finally (optionally) they are in the remote repository, e.g. GitHub, once the commit is published.



But there is also the branching and merging feature. Say we have a program working nicely but we want to add a feature. Before making changes to the code, we can branch off (git will swap the files in the wd with copy files) and make changes on the new branch, performing as many commits as we like. What's more, we can have many branches at the same time, each implementing a new feature. When a feature is fully implemented and tested, we can merge its branch with the master branch. Note when we switch branches, Git will automatically switch all the files in the wd with those whose changes belong to the branch (the original files, as well as files from all branches, are retained by Git in a hidden directory)

Comparing changes across stages (and through time)

We can use the `git diff` command to inspect differences between files. This section reviews the basic options. We will use the difference inspection GUI tool developed by p4merge. Instructions on how to download and install this software is found in the appendix. Git can be configured to use the p4merge tool by default when using the command `git difftool`

Note that `git diff[tool]` is repository-based by default – it will show changes made to all tracked files. If only a certain file is of concern, use a command like `git difftool filename` which would only show changes to filename between the wd and the staging area. There are virtually unlimited ways to compare versions, between commits, branches, stages, etc., but the below usages are most common and show the typical command pattern.

git difftool [filename]	Compares wd and staging area
git difftool HEAD [filename]	Compares wd and last commit
git difftool --staged [filename]	Compares staging area and last commit
git difftool commitID1 commitID2	Compares two commits
git difftool commitID filename	Compares some commit with some file
git difftool commitID --staged	Compares some commit and staging area
git difftool branch1 branch2	Compares last commits between branches
git difftool tag1 tag2	Compares two tags

Basic Git Commands

git init [*projname*]	initiate a new git project (give a name only if you want to make a new, empty directory, else just use the command in the existing project directory which may or may not contain files already)
git status	show the stages of changes to files
git commit -m "*message*"	commit staged changes and type a message inline
rm -rf .git	remove all traces of Git from a project (this is obv not a Git command)
git clone http://*url*	Initiate a git project from an existing remote repository e.g. a project's GitHub. Note you probably have to fork a GitHub repo to your personal account before being able to clone it (fork is a button to click on a GitHub project page, not part of the Git CLI).
git pull origin master	This is a best practice before pushing to a remote repo. If any changes were pushed to the remote repo between when it was cloned and now, this will allow you to merge any potential conflicts before pushing your changes to the remote repo.
git push origin master	publish any new commits in the local repo to the remote repo. "Origin" is just a conventional name for the remote repo and "master" signifies that you are pushing to the master branch
texteditor ~/.gitconfig	open the config file in *texteditor* e.g. nano

Basic Git Commands 2

git commit -am “*message*”	commit changes directly from the working directory, bypassing the staging area, and give a message
git ls-files	show files being tracked by git
git mv *file1 file2*	Rename or move a file in a git-tracked directory. (if you just use the bash command `mv`, git will register this as a deletion of *file1* and a new file, *file2*.)
git rm *file*	Delete a file. (this fast tracks the change directly to the staging area.)
git log	Show full commit history. There are *many* options for this command, type `git help log` to see them.
git log --oneline --graph --decorate	See the commit history in a more abbreviated, visual format
git log --follow -- path/to/file	See the commit history for a single file.

Git Aliases

Git allows the user to save verbose git command lines under an alias, making it easier to use the command. Using the git CLI itself, we can add aliases to the .gitconfig file.

`git config --global alias.*aliasname* "*full command omitting git*"`

The "global" indicates that we want to make the alias available at the user level, i.e. in all local repositories.

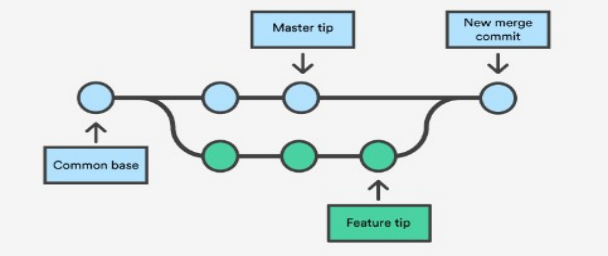
The .gitignore file

Git provides the facility to ignore certain files (regardless if they are in the repository and undergo changes). Git uses a text file called .gitignore located in the wd of the repo, to indicate which files are to be ignored. The format of .gitignote is one expression per line, an expression typically being the name of a file to be ignored, or a pattern e.g. "*.ipynb". If you want to specify that an entire directory in the repository should be ignored, add the following line to .gitignore: "*directoryname*/". The .gitignore file itself should be tracked for version control.

Git Concepts

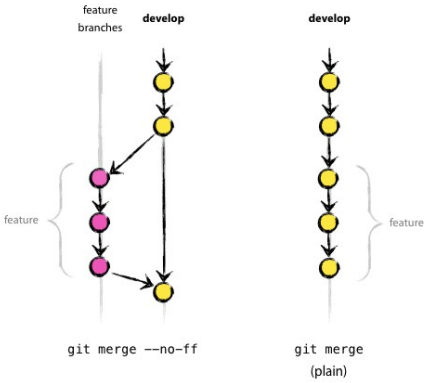
HEAD – a pointer or reference to the last commit on the current local branch
Master – a naming convention for the repo's main, default branch
Origin – shorthand name for the remote repo from which the project was originally cloned
Origin/Master|HEAD – the corresponding objects of the remote repo

Git Branching



git [-r|-a] branch	show local | [remote|all] branches
git branch new_branch	make a new branch
git -m old_name new_name	rename a branch
git -d branch_name	delete a branch (can't currently be on the branch)
git checkout branch_name	change branches
git checkout -b new_branch	make new branch and switch to it

Checkout to the branch you wish to merge into,
then type `git merge new_feature_branch_name`



When merging a commit from a direct descendant, git is smart enough to do a fast-forward: it simply moves the branch pointer to the last commit of the branch you're trying to merge (no actual new commit is constructed). The `--no-ff` flag preserves the topology, for cases when it's important to be able to look at the actual history (a legitimate new commit is constructed, a "merge commit").

Occasionally git will not be able to complete an automatic merge after issuing the command `git merge branch_name`; there are one or more conflicts between the branches – someone has to go in there are settle them. This is the case when you get the message "Automatic merge failed; fix conflicts and then commit the result" while attempting to merge. One could edit the relevant files directly (but note the files will be marked up by git), or issue the command `**git mergetool**` (if configured, e.g. with p4merge) to use a GUI tool to fix the conflict(s). This means, you look at all the merge conflicts and decide what the affected files should actually look like, because both branches can't win. After fixing the conflicts, do a commit. Note that after there is a merge conflict, git goes into a state where further merges and commits are not allowed until the conflicts are fixed and a merge commit is executed.

Git Rebase

While working on a new feature branch, you (or another developer) may have committed changes to the master (or other source) branch. You may way to incorporate these changes into the new feature branch and keep working on the new feature. There are two scenarios: either there will be conflicts, or there won't be. Checkout into the new feature branch, and issue the following command:

```
git rebase source_branch_name
```

If there are no conflicts, the history on the new feature branch will essentially be "pushed forward", as if the branch had originated after the changes in the source branch took place and were committed. If there are conflicts, they can be fixed manually or with the help of a GUI (if configured) using the following command:

```
git mergetool
```

After fixing the conflicts, execute a commit on the new feature branch, and problem solved (you may have to follow the prompts given by the CLI). Another option is to abort the rebase by issuing the command:

```
git rebase --abort
```

Often times you'll want to rebase from a remote repo such as Github. If you simply type `git pull origin branch_name`, it's the same as a merge (from the remote repo), not a rebase, which may not be desirable. A better solution is to issue the command:

```
git pull --rebase origin branch_name
```

Git Stashing

Sometimes we'd like to save a snapshot of our work, without committing. For instance, we're working on a feature but we have to hot fix something already in production. We want to revert to the most recent commit and do the fix, but we don't want to lose our changes, nor commit them yet. Git implements a *stack* of stashes, which function essentially like snapshots.

git stash [-u] [save "message"]	stashes the current changes [including untracked files] [with a message] and reverts to last commit
git stash apply [stash@{#}]	go to the most recent stash | [a specific stash]
git stash list	list all the stashes
git stash drop	delete most recent stash
git stash pop	go to most recent stash, and remove it from the stack of stashes
git stash clear	delete all stashes
git stash branch new_branch	create a new branch, switch into it, apply the stash *from the top of the branch stack* to the branch, and then drop the stash

Git Tagging

Git provides the utility to make tags on commits – simple phrases e.g. “v-1.2.3” to show the significance of the commit. These can also be annotated, to contain as much information about the commit as you like.

git tag tag_name	makes a tag on the current commit
git tag -a tag_name [-m “annot”]	makes an annotated tag and opens the default text editor to write the annotation [makes the annotation a short message]
git tag --list	lists all tags
git tag --delete tag_name	deletes a tag
git show tag_name	shows info about a tagged commit
git tag tag_name 7_char_commit_id	makes a tag on a given commit
git tag tag_name -f 7_char_commit_id	switches an existing tag to a different commit

By default `git push origin branch_name` does not publish tags to the remote repo. In order to accomplish this, you could use the following commands:

git push origin tag_name	publish a given tag
git push origin master --tags	publish all tags

GitHub Pull Requests and Team Workflow

To make a change to master branch when collaborating with other devs, push the new feature branch to GitHub with command `git push origin new_branch_name` (the branch needn’t already exist in GitHub). Then go to GitHub, navigate to the new branch, and click on ‘pull request’; leave a message and click ‘create pull request’. This tells GitHub to attempt to merge master with the new branch. If there are conflicts, you can resolve them using the web interface provided by GitHub, and then mark them as resolved. Then, click on ‘merge pull request’, leave an optional message, and then ‘confirm merge’ to merge the new branch with master branch (usually, you don’t review and merge your own pull requests, but rather a team lead or peer does this). As an optional last step, delete the new feature branch from GitHub by clicking ‘delete branch’.

You can add commits to your pull request after you made them. Simply push to the same branch. This is useful for instance when a peer does not approve your changes, leaves a comment for you to fix something, so you fix it and do another commit. This is how a back and forth dialog goes.

Adding a Remote Repository

If you want to set a a Github repository as a local project’s remote repo, type the following:

git remote add origin https://github.com/user/repo.git

To verify the remote repo, type:

git remote -v

Appendix

Installing p4merge GUI tool

Go to the perforce website, click ’Downloads’, and select ‘Helix Visual Merge Tool (P4Merge)’, and then select your platform and click ‘Download’ (or `wget` the correct file). It should come down as something like ‘p4v.tgz’. Unpack it with `tar zxvf p4v.tgz`. Then type `sudo mkdir /opt/p4v` and then `cd` into the directory ‘pv4-timestamp’ that unpacked from tar, and then type `sudo mv * /opt/p4v` and then `sudo ln -s /opt/p4v/bin/p4merge /usr/local/bin/p4merge` to make a symbolic link.

To setup p4merge as the visual diff and merge tool for git, you can use commands that can be found easily online, or just type the following lines manually into the ~/.gitconfig file:

```
[merge]
        tool = p4merge
[diff]
        tool = p4merge
[mergetool "p4merge"]
        path = /usr/local/bin/p4merge
[difftool "p4merge"]
        path = /usr/local/bin/p4merge
[mergetool]
        prompt = false
        keepBackup = false
[difftool]
        prompt = false
```