**Git Cheat Sheet**

**Introduction**

The purpose of this cheat sheet is to help you with a few commands that you would very likely need to use to properly do revision control of your software.

**Getting Help**

If the command is complex and you can’t remember its syntax, type

git <*command*> –h

for help display in the console. Or you may type

git help <*command*>

to get a more complete help in HTML. E.g.,

git commit –h

or,

git help commit

**Get Your Code Base Revisioned Controlled by Git**

To start revision control your codebase, you need to get into the root of the code base, and then type:

git init

This lets Git prepares the codebase to be revision controlled. It creates a new folder called .git in the root of the codebase that keeps a history of all reversions that will be created.

**Get The Status of The Working Tree**

A very handy Git command is to find out the status of the current working tree against the local repository. This is done through:

git status

This command reports *untracked* files if the code-base happens to contain some files before issuing git init. All files that are in working tree but not in local repository are treated as *untracked* files.

**Tracking Files**

To track files, Git needs to be told through the following command:

git add <*path-name*>/<*filename*>

e.g.,

git add src/main.c

This adds the file main.c in src folder to *staging area*. By adding files to staging area, they are being tracked. A staging area is a place to assemble all files before sending to the local repository.

**Untracking/Unstaging Files**

Sometime files maybe accidentally added for tracking (and subsequently transferred to the staging area/index) which were not intended to. They can be removed by executing the following command:

git reset <*path-name*>/<*filename*>

e.g.,

git reset src/main.c

Now the file becomes *untracked* again. This process is known as *unstaging*.

**Deleting Files**

Sometime a file is not needed anymore and ready to be thrown away. It can be removed using the following command:

git rm <*path-name*>/<*filename*>

e.g.,

git rm src/main.c

This will cause the file to be deleted and Git automatically adds it into staging area. Another way to remove a file is by issuing a \*nix command (not related to Git):

rm <*path-name*>/<*filename*>

e.g.,

rm src/main.c

This step does not cause Git to automatically stage the deleted file (and therefore will not be ready for a commit). To force Git to stage it, issue the former command. In this regard, the latter command is redundant and is best to avoid.

**Reverting Deleted Files**

If files were accidentally removed, it can be recovered by issuing the following command:

git checkout -- <*path-name*>/<*filename*>

e.g.,

git checkout -- src/main.c

if the **file has not been staged**. Otherwise it needs to be unstaged first before the command can take effect. See Untracking/Unstaging Files section on how to unstage a file. Note that the recovered file is as is when it was last committed. Whatever changes done after the last commit until the file was deleted is lost. So be careful when deleting a file.

**Commiting Files**

Before committing to local repository is possible, one needs to tell Git who the committer is so that it can be given credit (if those changes lead to good software) or blamed (if those changes break it) and the contact email. This is done typing:

git config –-global user.name “Max Paine”

git config –-global user.email “noname@jobless.com”

It is first-time setup and need no more configurations thereafter, unless email or name changed. Now committing files is allowed.

Once files are tracked/staged, they are ready to be sent to the local repository, a process known as *commit*. To do that, type:

git commit –m “Add a new file”

The option –m tells Git to attach the commit with the message “Adding a new file” stating what has been done. The message is a must so that we have a history of what have been done on the code-base in the local repository. Note that only files in the staging area will be committed. All untracked files are not.

**Remove Committed Files**

Committed files can be removed by issuing the following command:

git reset HEAD^

The HEAD is current active branch in the local repository. The ^ indicates that last commit is to be removed. To remove multiple recent commits type the following command:

git reset HEAD~2

The ~2 indicates removing the last 2 commits. **Warning: use this command with cautious since all removed commits are irrecoverable**.

**Pushing To Remote Repository**

There are several ways to push data to remote repository, but here only a couple is shown.

A remote repository can reside on the same machine as the local repository. This is introduced first because it is the simplest and requires few steps to setup. Suppose the target remote repository resides on G:\Projects\MP3Player.git, then that folder must be initialized before it is usable. To do that, get into the folder and type:

git –-bare init

This performs the same initialization as before except that the .git folder is not created. The content of .git is left exposed in MP3Player.git folder instead.

Now return to the code-base folder and type:

git remote add <*alias*> <*URL*>

e.g.,

git remote add RemoteRepo file:///g/Projects/MP3Player.git

This creates an alias for the remote repository, so that the long URL name can be referenced using the shorter alias name.

To push the local repository to remote, type:

git push RemoteRepo master

which means push the content of branch in the local repository name master to the remote URL. This command will not sync the local and remote copies. If they differ (after some commits on either repository), Git will not inform. To request Git to do otherwise, issue the following command instead:

git push –u RemoteRepo master

The option –u tells Git to sync between local and remote repositories—a process known as *remote tracking*. If they differ, Git will inform whether the remote is lagging by a few commits or it has diverged against the local working tree when the following command is issued:

git status

This can be useful as it eases the burden of having to remember if the remote is up-to-date or not. Once in tracking mode, the branch can be pushed simply with:

git push

without needing any parameters since Git now knows which remote branch to push to.

As a side-effect, the remote branch name is added into the branch list (Branch list can be viewed by typing “git branch –a”). The name is suffixed with remotes/.

NOTE: A (file) remote access error can happen when pushing an update. The most probable reason is because the storage device is not there, especially if the remote repository resides on an external drive/pendrive. When this happens, the error will persist even the external drive/pendrive is made available. This is because Git bash is oblivious of the newly mounted device. To resolve the problem, close ALL Git bash windows and then start a fresh Git bash. Now it should see the device. Another possible reason is because the remote repository folder has been relocated to somewhere else. To re-correct this, modify the alias to the relocated URL.

**Finding Out Remote Aliases**

Remote alias is a convenient way of referencing a long URL of a remote repository. Alias can be created as many as one likes. To find out all aliases that have been created, type:

git remote -v

**Pushing To Remote Repository With Different Remote Branch Name Than Local**

Sometimes there is a need to push from local branch to the remote repository with a different remote branch name. This can be easily done through the following command:

git push <*alias*> <*local-branch-name*>:<*remote-branch-name*>

e.g.,

git push RemoteRepo master:new\_name

which pushes local *master* branch to remote *new\_name* branch.

**Fetching and Pulling Remote Repository**

There can be times where the remote repository is also updated by other peers. This would cause the local to be out of sync from the remote. Since Git running on the local machine is a passive program, it will not know that until the following command is issued:

git fetch

which only fetches from the remote to local. Since the remote copy has diverged from the working tree copy, the differences need to be merged to the working tree. To do that, type:

git pull

which pulls form the local and merge to working tree. Often the merging process is automatic, but at times it can fail. When fail, manual merging is required.

After successful merging, the copy on the working tree and the remote is the same. However the local copy is not and need to be updated. This can be done using the usual commit command:

git commit –m “Merge remote copy with local copy”

**Cloning Remote Repository**

To clone a fresh remote repository on a local machine, type:

git clone <*URL*>

e.g.,

git clone file:///g/Projects/MP3Player.git

which retrieves the remote repository and install it fresh under the folder where Git command is issued. The folder name of the local repository is the same as the remote (in this example it is MP3Player.git/). To use a different name, say MyPlayer, type:

git clone <*URL*> <*local-folder-name*>

e.g.,

git clone file:///g/Projects/MP3Player.git MyPlayer

This step clones the *master* branch of the remote repository. To clone other branch, type

git clone –b <*remote-branch-name*> <*URL*>

e.g.,

git clone –b NewFeature file:///g/Projects/MPEGPlayer.git

**Merging Branches**

It is a good practice to branch out from the master copy so that a new software feature can be developed without messing up the master branch. At some point when the feature is stable, it is very likely to introduce into the master branch. This process is called *merging*. To merge changes done on other branch to master, first checkout the master branch by issuing:

git checkout master

Then issue,

git merge <*other-branch-name*>

e.g.,

git merge NewFeature

Normally, the merging process is automatic. However, at times failure can happen where manual merging is required to resolve conflict. The files of un-resolve conflict are given.

**Reverting To Previous Commit**

Revert

git revert <*SHA-1*>

**Creating a New Branch**

To create a new branch, type:

git branch <*new-branch-name*>

e.g.,

git branch development

which clone the current branch and name it development.

**Changing Branch**

Changing branch is as easy as typing:

git checkout <*existing-branch-name*>

e.g.,

git checkout development

**Checking Out a Commit**

Sometimes it is desired to browse through the files in the previous commit. This can be done by issuing:

git checkout <*SHA-1*>

e.g.,

git checkout f417b4cd417be5d7bd4afaf7457b4bbb06445509

**Checking Out a Remote Branch to Local Repository**

**[This info may not be valid. Please double check]**

If there is a remote branch not available on local repository, it can be easily made available by checking out the remote branch:

git checkout <*remote-branch-name*>

e.g.,

git checkout NewFeature

A new branch with the same name, NewFeature, will be created and will track the remote branch.

**Show Commit Logs**

To view the commit history, type:

git log

To view only the few recent commits, use the following command:

git log -<*number\_of\_commit*>

e.g.,

git log -2

The option -2 indicates to Git to display only 2 recent commits.

**Show Commit Logs With Files Compare**

To view the commits and the changes made to the commits, type

git log –p

To limit the number of commits and the display of their changes, type:

git log –p -4

**Displaying the Contributors & Their Commits**

Git can display all contributors and their commits (in brief). To do that type:

git shortlog

Type git shortlog -s to just display the name of contributors and their respective number of commits done.

**Ignore Some Files And Not Tracked By Git**

List all ignore directories and files in *.gitignore* file and place it in the codebase directory.

For example, to ignore everything in build folder, place the following in the file:

build/\*

Suppose there is a file (say README.txt) in the ignored folder that shouldn’t be ignored, then place the following in the file:

build/\*

!build/README.txt

The first line says, ignore all files and folders under build folder. The second line says, but don’t ignore README.txt in it.

Note that if the repository has already contained the files/folders to be ignored, then they will be retained even *.gitignore* says otherwise. To remove them from the repository, use the following command:

git rm –r --cached *<files>*

The option –r means execute the command recursive. This command is dangerous because it will delete files. To be safe, always run with run-dry option to see what will be deleted first:

git rm –r –n --cached *<files>*

e.g.,

git rm –r –n –cached build/.

will run-dry the recursive remove of all files/folders under build folder.

**Deleting Local/Remote Branch**

Deleting local branch can be done through the following command:

git branch –d NewFeature

Deleting remote branch can be done through the following command:

git push <*remote-name*> :<*remote-branch-name*>

e.g.,

git push RemoteRepo :NewFeature

Take note of the space between <*remote-name*> and the “:”.

**Tracking Remote Branch by a Local Branch**

To track a remote branch by a local branch, type the following:

git branch –u <*remote-alias*>/<*remote-branch*> <*local-branch*>

e.g.,

git branch –u origin/GreatFeature NewFeature

If the local branch intended to be used to track the remote is the currently checked-out branch, then the <*local-branch*> field can be ignored. If the command gives the following error:

error: the requested upstream branch 'origin/master2' does not exist

it is because the remote branch is not in the branch list yet. To include it, issue:

git fetch <*remote-alias*>

e.g.,

git fetch origin

The easiest way is to issue a push command, i.e.:

git push –u <*remote-alias*> <local-branch>

e.g.,

git push –u RemoteRepo master

But this will update the remote as well (see Pushing to Remote Repository section as well).

**Untracking Remote Branch**

To stop tracking a remote branch, issue the following 2 commands:

git config --unset branch.<*branch-name*>.remote

git config --unset branch.<*branch-name*>.merge

e.g.,

git config --unset branch.NewFeature.remote

git config --unset branch.NewFeature.merge

NOTE: This method is probably a hack since it manually removes tracking information in the configuration file. I don’t know an alternative way.

This step does not remove the remote branch from the branch list (which was created when remote tracking was established) however. It is not a big deal since it does not affect anything, but if the appearance of the name is annoying, it can be removed by issuing:

git branch –d -r <*remote-alias*>/<*remote-branch*>

e.g.,

git branch –d –r origin/GreatFeature

This DOES NOT remove the remote branch from the remote repository. It only removes remote branch from the branch list. (Note: To view branch list, type “git branch –a”).

**Information About Remote Repository**

Valuable information such as about remote repository, remote branches, and which local branches are tracking which remote branches, can be found by issuing the following command:

git remote show <*remote-alias*>

e.g.,

git remote show ORIGIN

**Finding Out Who Wrote/Modified the Code**

Git can keep track of who wrote and when for each line of code. To display the history, type:

git blame -w <*filename*>

e.g.,

git blame –w foo.c

**See the Difference Between Revision Files**

Git can help display the differences between similar files in different revisions. To do that type:

git diff <commit> <commit> -- <*filename*>

e.g.,

git diff HEAD HEAD~ -- foo.c

To compare with the current working tree, type:

git diff <commit> -- <*filename*>

e.g.,

git diff HEAD -- foo.c

If --<*filename*> is omitted, Git will display differences of all affected files.

Not that:

1. git diff - Compare index (staging area) with working tree
2. git diff HEAD - Compare local repository with working tree
3. git diff --cached - Compare local repository with the index

**Amending Date of the Latest Commit**

Git can amend the date of the last commit. Just issue the following command:

~~GIT\_COMMITTER\_DATE="Mon Jul 2 00:30 2013 +0800"~~

git commit --amend --date "Mon Jul 2 00:30 2013 +0800"

**Listing All Ignored Files**

Git can list all ignored files(?). Just type:

git ls-files --others -i --exclude-standard

Git can also list all ignored files specified in some specific file. Just type:

git ls-files --others -i --exclude-from=<*path-name*>/<*filename*>

e.g.,

git ls-files --others -i --exclude-from=.git/info/exclude

will list all ignored files stated in .git/info/exclude file.

**Setting Remote HEAD to Other Remote Branch**

When you type

git branch –a

you will get a list of all local and remote branches, like:

\*development

master

remotes/origin/HEAD -> origin/master

remotes/origin/development

remotes/origin/master

Note that remotes/origin/HEAD points to origin/master, which means when you clone from remote repository, the origin/master is the branch that is checkout. But sometimes we may want to checkout other remote branch than remote master. To change the remote HEAD, type the following:

git remote set-head <*remote-alias*> <*remote-branch*>

e.g.,

git remote set-head origin development

**Lising All Remote Branches**

The command

git branch –r

shows all locally tracked remote branches. Non-locally tracked remote branches will not show up. To list all remote branches (locally tracked or not), type:

git ls-remote --heads <*remote-alias*>

e.g.,

git ls-remote --heads origin

**Forcing a Push to Remote Repository (Rewriting Remote Repositoty)**

At times, we may want to overwrite the remote repository with a fresh new codebase in the local working directory. To do that, type:

git push *<remote-alias> <local-branch>* --force

e.g.,

git push origin master --force

If the local master is tracked by the remote repository, then

git push --force

e.g.,

git push --force

will do. **Use this command with care as the overwritten remote repository is irrecoverable.**

As of the [latest 1.8.5 git release](https://raw.github.com/git/git/master/Documentation/RelNotes/1.8.5.txt), you can use

git push --force-with-lease

which will produce an error and not push if the remote was modified since your last fetch.

**View specific file in a revision**

git show HEAD~4:index.html

**Show branches graphically**

git log --pretty=oneline --graph --all

<http://gitready.com/intermediate/2009/04/16/find-unmerged-commits.html>

**Overwrite the current, or restoring, file/directory from specific revision**

*# Pull the whole bin folder from the previous revision*

git checkout HEAD~1 -- bin/

*# Pull the whole bin folder from the previous revision*

*# of the 'test' branch*

git checkout test~1 -- bin/

*# Pull the file.txt folder from the previous revision*

git checkout HEAD~1 -- bin/file.txt

*# Pull the file.txt folder from the 261dfab... revision*

git checkout 261dfab -- bin/file.txt

<http://gitready.com/intermediate/2009/03/18/restoring-a-directory-from-history.html>

**Reverting to previous revision**

You can revert to previous revision by typing:

git checkout <*SHA-1*> .

e.g.

git checkout f417b4cd417be5d .

or,

git checkout HEAD~4 .

Then commit with ‘git commit –m "Revert back to f417b4cd417be5d"’.

**Reverting the file/directory overwrite, or restore of the previous step**

The previous step does not only pull file/directory, but also automatically added into the staging area. It must be removed first. To do that:

*# Remove whole bin/ folder from staging area*

git reset HEAD bin/

*# Remove bin/file.txt from staging area*

git reset HEAD bin/file.txt

Now, if you 'git status', you will see the file/folder is either modified or untracked.

If it has been modified, you can now checkout the original copy from local repository, by:

*# Restore the move whole bin/ folder from staging area*

git checkout -- bin/

*# Restore the bin/file.txt file from staging area*

git checkout -- bin/file.txt

If it is untracked, you can now remove it, by:

*# Remove the bin/file.txt file from working directory*

rm -r bin/file.txt