Contents

1 Fast-Ref

`git add *`   stage all changed files

`git commit -am 'msg'`

`git commit - -amend`   Replaces last commit to include the updates & changes the commit msg

1.1 unstage

01 `git restore - -staged <file1, file2>`   Un-stages files 1,2

1.2 merge branches

01 `git merge <master> <branch>`   merge <branch> to master

02 `git merge <branch1> <branch2>`   merges: <branch2> to <branch1>

1.3 merge to master

01 `git branch`   Lists branches

02 `git checkout master`   switch to master

03 `git merge <branch>`   merge <branch> to master

04 `git log`

1.4 del a branch

01 `git branch`   Lists branches

02 `git checkout master`   switch to master

03 `git branch -d <branch>`   Deletes branch

04 `git push origin`

1.5 Rename a branch

01 `git branch -m <name>`   rename the current branch to <name>

`p`	(going back to HEAD)
`git reset -hard HEAD`∧	(going back to the commit before HEAD)
`git reset -hard HEAD`∼ 1	(equivalent to "∧")

2 First-Time Git Setup

First-time set-up & configuration for a new Git installation.

2.1 `git config`

`git config - -global user.name <FirstName LastName>`	Drop "- -global" option from these commands
`git config - -global user.email <email@example.com>`	to recognize you only locally.

`git config - -global core.editor <emacs>`	To use a different text editor (from system default) for git.
`git config - -list - -show-origin`	Shows settings and where they are coming from
`git config - -list`	Checking the Settings that are in effect.

`git config user.name`	Shows who it is configured to
`git config user.email`	Shows the email associated to git

3 log

3.1 GUI

`gitk`	Opens a visual commit browser (some GUI)

3.2 `git help`

`git --version`

`git help | git --help`

`git help <command>`

Gives help about <command> `git <command> --help`

Checks installed Git's version

Shows git help

3.3 `git status`

`git status`	Checks the current state of repo
`git status <file>`	Checks state of specific file

3.4 `log`

`git reflog`	Views the history of checkouts
`git log`	Shows a detailed log of commit history
`git log --oneline`	Shows only <commentsID> commit-comments
`git shortlog`	For a shorter log of commit history
`git shortlog -s`	Creates even much shorter log
`git shortlog -1`	Shows only the last 'one' commit

1. *-1*:

2. *-p*: shows the line diff for each commit

3. *-p –word-diff*: shows the word diff for each commit

4. *–stat*: shows stats instead of diff details

5. *–name-status*: shows a simpler version of stat

4 Create repo.

Two ways to create a repository:

4.1 `git init`

`git init <dir>`	Create new (empty) / reinitialize existing repo in <dir>
`git init`	Executing this in project-directory <dir> makes it a repo

4.2 `clone`

» `git clone <repo-url>`   It is the URL to a remote git repository <repo>

1. Creates a local `<repo>` folder & Initializes it as a git repository

3. Copies (pull-downs) all data from `repo-url` to the local folder

4. Automatically configures <repo> to point to the `<repo-url>`

5. `Checkout` to the local working directory

Once making change-&-committing files, one can `git push` the changes to the remote repository at <repo-url>

» `git clone <repo-url> <folder-name>`   Same as above but local repo can be <folder-Name> (different from the remote one)

» `git clone <repo1> <repo2>` Copies a local repo-folder to a new local folder

Delete: To remove git control delete ".git" from working director.

5 Staging

Staged-files are ready to be committed

5.1 `add`

`git add <file1 file2 file3>`	Adds files 1–3 to staging area
`git add *.txt`	stages all text files
`git add .`	stages all text files
`git add -A | git add --a[ll]`	Adds evergitything in and beneath
`git add --u`	Adds modified files (but not the new ones)

5.2 `.gitignore`

A hidden file (`.gitignore.txt`) in the root of repository specifies the files we do not need to keep track of their changes (e.g. *.exe files). Note that the files already tracked by Git are not affected. To remove all files from the repository and adding them back according to the rules in .gitignore:

Use: `git rm -rf --cached .` $\longrightarrow$ `git add .`   (see Appendix C)

6   Un-staging

6.1 `restore / rm`

`git restore --staged .`	Un-stages all the staged files
`git restore -S .`	Shorten form
`git rm -rf --cached .`	Un-stages all staged files <mark>+</mark>
`git restore --staged <file1, file2>`	Un-stages files 1,2
`git rm --cached <file>`	Un-stages `<file>`

7   Delete

7.1 `rm`

`git rm <files>`	Deletes from both the working directory and staged area.
	Note: you may lose all the changes (even the good ones):

8   Commit

`git commit`	Commits staged files & asks for "comment message" to the commit
`git commit -m 'msg'`	Commits & (`switch '-m'`) simultaneously adds a commit message
`git commit --all [|-a]`	Commits all the file in the staged area and asks for the comment
`git commit -am 'msg'`	Adds modified files to stage, commits them, and adds commit message

Un-commit

`git commit --amend -m "new-msg"`	Replaces the last commit of the current branch with the current staged files and replace its commit-msg with the "new-msg"; as last commit never happened.
`git commit --amend`	If no changes since last commit (e.g. immediately after a commit), This only changes the commit-msg. It opens the last commit-message in the editor for editing, any change overwrites last commit-msg.

Reset Author

`git commit --amend --reset --author`	Amends commit author & author date to the committer
`git commit --amend --author="Author Name <email>"`	Amends commit author with given author name & email, author date remains unchanged
`git commit --amend --date="2021-04-13T14:59:10"`	Amends the commit date (use ISO 8601 format for convenience)

9 `branch`

To create, rename, delete, etc. of a branch

`git branch`	Shows (list) both local branches
`git branch <name>`	Create a new (local) branch from the current Head (i.e. last commit)
`git branch -m <new-name>`	Rename the current branch to "new-name"
`git branch -d <name>`	Delete this branch, This do not delete if branch has unmerged changes
`git branch -D <branch-name>`	Force delete this branch, even if it has unmerged changes
`git branch --a`	Shows (list) both remote & local tracking branches
`git branch --r`	Shows remote tracking branches

10 Branch - `checkout`

Takes to branch

`git checkout -b <name>`	Creates a new-branch <name> from current Head & then, checkout to it ` + `
`git checkout master`	Switches to branch "master" (github)
`git checkout <branch-name>`	Switches to <branch-name> ` + `
`git checkout <remote-branch>`	
`git checkout -- <files>`	Discards all the changes in <files> and restore it from the staged-version
`git checkout --detach`	Detaches Head from current branch

"**checkout**" is the act of switching between different versions of a target entity.

11 `diff`: compare

Compares and shows differences between two instances.

`git diff`	Compares and shows differences between Working-Directory vs. Staging-Area
`git diff <file>`	Compares and shows modifications in current `file` in working-directory compared with `file` in staging-area
`git diff --staged`	Compares all the files in staging area with the last committed ones
`git diff HEAD`	Working-Area vs Last-Commit
`git diff <branch-name>`	Shows diff between branch and working-directory
`git diff <branch-1> <branch-2>`	Shows diff between two branches

12 Cleaning Working-Directory

`git clean --n`	See what would be done by `git clean` command (Dry-Run of clean Operation)
`git clean --1`	Interactively(?) removes un-tracked files from repository (Remove un-traced files)

```
git restore --staged .
Or simply you can

git restore -S .
```

13 `git reset`

13.1 Roll-back to a previous commit

`git reset --hard HEAD`	Reverts working copy to the HEAD (most recent commit) ` + `

`git reflog`	Shows current commit history or use `git log --oneline`
`git reset <commitId>`	Resets master to the commit <commitID> (absolute address). e.g., commit 0766c053c0ea2035e90f504928f8df3c9363b8bd
`git reset current~2`	Resets master to 2 commit before the current commit (relative address)

`git reset`	1. Removes everything from staging area
	2. Resets every modified files in working-space to the latest commit
	3. Brings them back to the working-area
	Note: you may lose all the changes (even the good ones):
`git reset <files>`	Un-stages "`<files>`" from the indexing → Reset to the latest commit → Leaves them in the working-area
`git reset path/to/file`	Un-stages files in "`path/to/file`" folder from the indexing, ... as above

13.2 `git reset HEAD`

`git reset HEAD - <files>`	Un-stages "`<files>`" from the index
`git reset HEAD -- path/to/file`	Un-stages files in "`path/to/file`" folder from the index
`git reset HEAD -- .`	Un-stages from the indexing all the files recursively and so forth to the subfolders

14 Remote Repository

14.1 Git-Hub

Given you have a GitHub account:

1. log-in to: https://github.com
2. Create a remote repository in https://github.com/yourgit/proj.git
3. `git remote add origin https://github.../proj.git`

15 Stash

A practical guide to using the git stash command

15.1 Removing Remote URL

`git remote -v`	Views the current remote
`git remote rm`	Removes a remote URL from your repository
`git remote rm master`	

15.2 Pushing

`git push -u origin master`	Sends local changes to remote repository (*origin*)

15.3 Pulling

`git pull origin master`	Pull-down any new changes (by collaborators etc.) from the remote repo.

15.4 Example: working with git and GitHub

1. `mkdir D:/proj`
2. `echo "# main.tex" » D:/proj/README.txt`
3. `cd D:/proj`
4. `GIT init`
5. `git add README.txt`
6. `git commit -m "first commit"`
7. `remote add origin https://github.com/BehN/Git-Help-LaTeX.git`
8. `git push -u origin master`

A Concept and definitions

Local repo. consists of following 3 "trees" maintained by git.

1. `Working Directory` holds the actual files.

2. `Index = staging area`.

3. `Head` = last commit.

Courtesy of https://rogerdudler.github.io/git-guide/

B Appendix: Reviewing some concepts with a descriptive style

Differences:

Uh oh, looks like there have been some additions and changes to the cat family. Let's take a look at what is different from our last commit by using the `git diff` command. In this case we want the diff of our most recent commit, which we can refer to using the HEAD pointer.

`git diff HEAD`

Staged Differences:

Another great use for diff is looking at changes within files that have already been staged. Remember, staged files are files we have told git that are ready to be committed. Let's use git add to stage octofamilyoctodog.txt, which I just added to the family for you.

`git add octofamily/octodog.txt`

Good, now go ahead and run git diff with the –staged option to see the changes you just staged. You should see that octodog.txt was created.

`git diff -staged`

Resetting the Stage:

So now that octodog is part of the family, cat is all depressed. Since we love cat more than octodog, we'll turn his frown around by removing octodog.txt.

You can unstage files by using the git reset command. Go ahead and remove octofamily/octodog.txt.

`git reset octofamily/octodog.txt`

Undo:

git reset did a great job of unstaging octodog.txt, but you'll notice that he's still there. He's just not staged anymore. It would be great if we could go back to how things were before octodog came around and ruined the party.

Files can be changed back to how they were at the last commit by using the command: git checkout – <target>. Go ahead and get rid of all the changes since the last commit for cat.txt

`git checkout - cat.txt`

Removing:

Ok, so you're in the clean_up branch. You can finally remove all those pesky cats by using the git rm command which will not only remove the actual files from disk, but will also stage the removal of the files for us.

You're going to want to use a wildcard again to get all the cats in one sweep, go ahead and run:

`git rm '*.txt'`

Removing one file is great and all, but what if you want to remove an entire folder? You can use the recursive option on git rm:

`git rm -r folder_of_cats`

This will recursively remove all folders and files from the given directory.

Committing Branch Changes:

Now that you've removed all the cats you'll need to commit your changes. Feel free to run git status to check the changes you're about to commit.

`git commit -m "Remove all the cats"`

Switching Back to master: Great, you're almost finished with the cat... er the bug fix, you just need to switch back to the master branch so you can copy (or merge) your changes from the clean_up branch back into the master branch.

Go ahead and checkout the master branch:

`git checkout master`

Preparing to Merge:

Alright, the moment has come when you have to merge your changes from the clean-up branch into the master branch. Take a deep breath, it's not that scary.

We're already on the master branch, so we just need to tell Git to merge the clean_up branch into it:

`git merge clean_up`

Keeping Things Clean:

You just accomplished your first successful bugfix and merge. All that's left to do is clean up after yourself. Since you're done with the clean_up branch you don't need it anymore.

You can use git branch -d <branch name> to delete a branch. Go ahead and delete the clean_up branch now:

`git branch -d clean_up`

git merge:

When you're done working on a branch, you can merge your changes back to the master branch, which is visible to all collaborators. git merge cats would take all the changes you made to the "cats" branch and add them to the master.

`git merge`

C   More about `.gitignore`

In each line of this file, it specifies the (type of) files that we do not want to stage (never) and git should ignore when staging the files with `--all` switch.

Useful "pattern format" to specify a group of files/folders.

1. "`blank line`": matches no files, so it can serve as a separator for readability

2. "`#`": makes it a comment line; `\#` for the patterns literally leading with `#`

3. "`!`": ignore but excluding the matching pattern; `\` for the patterns literally leading with
   e.g.: `important!.txt`: ingores all `important*.*` except the ones with `.txt` extension

4. An asterisk "`*`" matches anything except "`\`".

5. The character "`?`" matches any one character except "`\`".

6. "`**/<this>`": match `<this>` in all directories. `e.g.`:

   - "`**/foo`": matches file or folder "`foo`" anywhere.

   - "`**/foo/bar`": matches file/folder "`bar`" in/under "`foo`" folder

7. "`<folder>/**`": match everything inside "`<this>`" and all the paths underneath it

8. "`/**/`": matches zero or more directories. `e.g.`:

   - "`a/**/b`" matches "`a/b`", "`a/x/b`", "`a/x/y/b`" and so on.

9. "`*`": Usage

   - "`hello.*`": matches any file or directory whose name begins with "`hello.`"

   - "`/*`": matches any file or directory

   - "`/foo/*`": matches any file or directory inside "`/foo`" and folders underneath

10. The forward-slash "`/`", all paths are relative from the `.gitignore` file location

    - "`a/**/b`" matches "`a/b`", "`a/x/b`", "`a/x/y/b`" and so on.

    - "`foo/`": matches a directory "`foo`" and paths underneath it, but does not match a regular file or a symbolic link foo

    - "`doc/frotz/`": matches folder "`./doc/frotz`", but not "`a/doc/frotz`" directory

    - "`frotz/`": matches both folders as "`./frotz/`" and "`a/frotz`"

    - "`doc/frotz`": and "`/doc/frotz`" have the same effect. A leading slash is not relevant when a middle slash in the pattern

    - "`foo/*`": matches "`foo/test.json`" (a regular file), "`foo/bar`" (a directory), but it does not match "`foo/bar/hello.c`" (a regular file), as the asterisk in the pattern does not match "bar/hello.c" which has a slash in it

11. "`a/**/b`": matches "`a/b`", "`a/x/b`", "`a/x/y/b`" and so on

Example:

```
$ cat .gitignore
# exclude everything except directory foo/bar
/*
!/foo
/foo/*
!/foo/bar
```

  - This exclude everything except folder "`foo/bar`"

- "**/***": Exclude everything (even files inside "**/foo**" and "**/foo/bar**"

- "**!/foo/bar**": keeps the empty folder "**/foo/bar**"

(note the /* - without the slash, the wildcard would also exclude everything within foo/bar):

C.1   Sample cmd file to Create .gitignore

This is a simple windows script (batch file) that can be used to generate a sample (and fairly complete) `.gitignore`. Both the following script and `.gitignore` from it can be edited to customize it with your need.

GitIgnore.cmd:

```
@call C:\SBN\!!fst\Srvr\Init>nul
::\#!\bib\csh

::————————————————————
set FN=..\.gitignore
attrib -h -r %FN%
del /s/q %FN%>nul
::————————————————————

:: Git
echo **/.gitignore>%FN%

:: WIN:
echo **/.dropbox>>%FN%
echo **/desktop.ini>>%FN%
echo **/.tmp>>%FN%
echo **/nul*>>%FN%
echo **/*.exe>>%FN%

:: IEEE
echo **/IEEEtran.bst>>%FN%

:: PDF
echo **/*.pdf>>%FN%
echo **/*.PDF>>%FN%

:: Matlab:
echo **/*.asv>>%FN%

:: Graphics
echo **/*.eps>>%FN%
echo **/*.png>>%FN%
echo **/*.jpg>>%FN%
echo **/*.jpeg>>%FN%

:: Hspice:
echo **/*.log>>%FN%
echo **/MIL.*>>%FN%
echo **/sxcmd.*>>%FN%
echo **/*.sx>>%FN%
echo **/*.lis>>%FN%
echo **/*.fsdef>>%FN%
echo **/*.str>>%FN%
echo **/*.ic0>>%FN%
echo **/*.st0>>%FN%
echo **/*.pa0>>%FN%
echo **/*.sw0>>%FN%
echo **/*.tr0>>%FN%
echo **/*.ac0>>%FN%

:: TexnicCenter:
echo **/*.out>>%FN%
echo **/*.aux>>%FN%
echo **/*.blg>>%FN%
echo **/*.bbl>>%FN%
echo **/*.toc>>%FN%
echo **/*.dvi>>%FN%
```

```
echo **/*.bak>>%FN%
echo **/*.prj>>%FN%
echo **/*.ppl>>%FN%
echo **/*.lot>>%FN%
echo **/*.lof>>%FN%
echo **/*.tps>>%FN%
echo **/*.synctex>>%FN%
echo **/*.tmp>>%FN%
echo **/*.tps>>%FN%
echo **/*.pdfsync>>%FN%
echo **/*.ps>>%FN%
echo **/*.undo>>%FN%
echo **/*.tex~>>%FN%
echo **/*.tex.backup>>%FN%
echo **/*.maf>>%FN%
echo **/*.mtc>>%FN%
echo **/*.mtc[0-9]>>%FN%
echo **/*.mtc??>>%FN%


::Vim:
echo **/*.project.vim>>%FN%
echo **/*.glg>>%FN%
echo **/*.glo>>%FN%
echo **/*.gls>>%FN%
echo **/*.ist>>%FN%
echo **/*.dcl>>%FN%

::TeXStudio/TeXMaker:
echo **/*.gz>>%FN%
echo **/*.spl>>%FN%
echo **/*.fls>>%FN%
echo **/*.brf>>%FN%
echo **/*.xml>>%FN%
echo **/*.bcf>>%FN%

::Beamer:
echo **/*.nav>>%FN%
echo **/*.snm>>%FN%

::XHTML:
echo **/*.idx>>%FN%
echo **/*.css>>%FN%
echo **/*.ilg>>%FN%
echo **/*.ind>>%FN%

::Others:
echo **/*._*>>%FN%
echo **/*.ini>>%FN%
echo **/*.fdb*>>%FN%

::Batch/cmd scripts:
echo !!.bit/**>>%FN%
echo **/*.bat>>%FN%
echo **/*.cmd>>%FN%

attrib +h +r %FN%
::————————————————————————————————
dir /AH /B %FN%

start notepad++ %FN%
call %IS%\end 5
```

D How to clone my webpage from GitHub

Copy the webpage's URL:

1. On GitHub $\longrightarrow$ Go to Repositories

2. Click on your "personal Web-Page"

3. Click on Clone/download

4. Right-Click+Copy the web URL (e.g.: ``https://github.com/personalWebPage.git``)

Make a local repository:

5. Go to a directory as e.g.: "D:\code"

6. Right-Click and select "Git Bash here!"

7. If the git installation (is new &) has not been initialized yet take the next two steps. This is only done once for the git installation!

   `git config -global user.name "John Doe"`

   `git config -global user.email you@email.com`

How to clone:

8. `git clone https://github.com/personalWebPage.git`

   N.B.: As a result, the directory "personalWebPage" is created containing a copy of git-repository. This new directory is not a git repo yet???? really???

How to branch:

9. `ls` $\longrightarrow$ `cd personalWebPage`

10. `checkout -b dev`   This makes `'dev'` which is a new local branch

11. `branch`   Shows existing branches, the active one is in green

12. `checkout dev`   switches git to new local branch `'dev'`

How to stage and commit: (After applying required editing!)

13. `git add --a`

14. `git status`   check status on branch `'dev'`

15. `git commit -m "your commit note goes here!"`

How to push the branch to GitHub:

16. On GitHub $\longrightarrow$`create pull request`

17. `push origin dev`   push `'dev'` to remote, creates if not exist