Git Configurations

```
git config --global user.name "[name]"
```

sets the name desired to be attached to your commit transactions

```
git config --global user.email "[email address]"
```

set the email address to be attached to your commit transactions

```
git config --global color.ui "[auto, true, or false]"
```

enables helpful coloriaztion of command line output

```
git config --global core.editor "[name of editor]"
git config --global core.editor.path "[editor path]"
```

sets the default editor

```
git config --global merge.tool "[name of merge tool]"
git config --global merge.tool.path "[merge tool path]"
```

sets the merge tool editor

```
git config --global differ.tool "[name of differ tool]"
git config --global differ.tool.path "[differ tool path]"
```

sets the differ tool editor

```
git config --global push.default current
```

whenever you create a new local branch, a remote branch will always be created. Afterwards use 'git push' to the current branch of the same name (its upstream branch).

```
git config --global push.default matching
```

whenever you use 'git push', Git will push local branches to remote branches that already exists with the same name

```
git config --global --list
```

list all Git global configuration settings

```
git config --list
```

list all Git configuration settings

Git Aliases Configurations

```
git config --global alias.br branch
```

```
git config --global alias.co checkout
git config --global alias.ci commit
git config --global alias.st "status -s -b"
git config --global alias.last "log -1 HEAD"
git config --global alias.unstage "reset HEAD --"
git config --global alias.log2 "log --pretty=format:'%h %s [%an]' --graph"
git config --global alias.history "log --all --graph --decorate --oneline"

git config --global alias.who2blame "log --graph --pretty=format:'%h %ad-%s
[%an]'"

git config --global alias.logg "log --graph --decorate --pretty=oneline --
abbrev-commit --all"

git config --global alias.history2color "log --pretty=format:'%Cgreen%h%Creset
%ai | %s %Cblue[%an] %Cred%d' --date=short -n 10 --coloralias.graph=log --
graph --decorate --pretty=oneline --abbrev-commit --all"

Git Initialization

To get start - create a new folder or directory
from the command line --

echo "#Some-new-app" >> README.md
git init
git add README.md

git commit -m "Initial commit"
git branch -M main
git remote add origin "[https://github.com/username/repository_name]"
git push -u origin main

or to push an existing repository
from the command line --

git remote add origin "[https://github.com/username/repository_name]"
git branch -M main
git push -u origin main

Git Clone Repository
```

git clone "[https://github.com/username/repository_name]"
creates a directory named of the git repository, initializes a .git folder inside it, pulls down all the data for the repository, and checks out a working copy of the latest version of the repository.

git clone "[https://github.com/username/repository_name]" "[folder name]"
does the same thing as the above git clone command, but the target folder is the "[folder name]".

Permanently Stop Tracking a File

(If a file is already tracked by Git, .gitignore file does not apply. Git will continue to track changes to that file)

1. Add the file to .gitignore file

2. Run the following command
git rm --cached "[fileName with extension]" -r

Git Workflow

git branch develop
git push -u origin develop

Branch develop will contain the complete hisory of the project, whereas branch main will contain a shorten version. Instead of branching off branch main, feature branches will use branch develop as their parent branch. When feature branch is complet, it gets merged back into branch develop. Feature branches should never interact directly with branch main.

git checkout develop
git checkout -b feature_branch

When done with the development work on the feature branch, the next step is to merge the feature_branch into branch develop.

git checkout develop
git merge feature_branch

Once branch develop has acquired enough features for a release. A release/1.X.X is merged into branch main. Then, to finish a release branch, use the following:

```
git checkout develop
git checkout -b release/0.X.X
```

```
git checkout main
git merge release/0.X.X
git tag -a v0.X.X -m "version 0.X.X"
git branch -D release/0.X.X
```

Maintenance or "hotfix" branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches, except they are based on the branch main instead of the branch develop. Hotfix branches are the only branches made directly from branch main.

```
git checkout main
git checkout -b hotfix_branch
```

When finishing a hotfix branch, it gets merged into both branch main and branch develop.

```
git checkout main
git merge hotfix_branch
git tag -a v.X.1 -m "version 0.X.1"
git checkout develop
git merge hotfix_branch
git branch -D hotfix_branch
```

Example of Complete Gitflow:

```
git checkout main
git checkout -b develop
git checkout -b feature_branch
#Work happens on feature branch
```

```
git checkout develop
git merge feature_branch
git checkout main
git merge develop
```

```
git branch -d feature_branch
#When ready for release

git checkout develop
git branch release/1.X.X
git checkout main
git merge release/1.X.X
git tag -a v1.X.X -m "version 1.X.X"
git branch -D release/1.X.X

Example of Complete Gitflow Hotfix:
git checkout main
git checkout -b hotfix_branch
#Work is completed to hotfix branch

git checkout develop
git merge hotfix_branch
git checkout main
git merge hotfix_branch
git checkout -b release/1.X.1
git tag -a v1.X.1 -m "version 1.X.1"
git branch -D release/1.X.1
git branch -D hotfix_branch

Git Pull Request

1. On Github, fork the repository into your Github account.

2. Create a local copy on your computer and create a branch.
from the command line --

git clone "[https://github.com/username/repository_name]"
cd "[repository_name]"
git checkout -b "[pull_request branch]"

3.  Make changes to the pull_request branch.

git add "[change_made]"
git commit -m "[Update: changes_made]"
git push --set-upstream "[pull-request branch]"
```

4. Create a pull request

Our changes are now uploaded to our remote Git repository. We are ready to make a pull request. To do so, go to the homepage of the forked version of the repository.

There is a button which says, "New pull request" that is associated with our new branch. When this is pressed, we create a pull request with our changes.

In the form we specify the name of our pull request (which is automatically set as a commit message of your most recent changes) and the description for our pull request.

Workflow for Changing a File or Folder

1. Update a file or folder in the Git repository

2. Use git add to add those changes to the staging area

3. Use git commit to move changes from the staging area to a commit

4. Use git push to move/add changes to the main repository

Git Add

git add "[filename and extension]"

add filename to the staging area

git add --all or git add .

add every change made to files and folders from our respository to the Git staging area

Git Commit

git commit

a text editor will be opened in which you can write a message

git commit -a

automatically stages all modified files to be commited in our repository

git commit -m "[Commit message]"

the -m flag stands for message and is used to directly write a brief message skipping over a text editor openning.

git commit --amend -m "[Amended commit message]"
amends our most recent previous commit message

Git Push

git push "[remote]" "[branch]"
creates a local branch in the destinated repository; and/or pushes the specified branch to along with all the necessary commits and internal objects

git push -u "[remote]" "[branch]"
when pushing a branch for the first time, this command will configure the relationship between the remote and local repository, so that you can use 'git pull' and 'git push' with no additional options in the future.

git push "[remote]" --all
pushes all of your local branches to the specific remote

git push "[remote]" --force
forces the git push even if it results in a non-fast forward merge. DO NOT USE THE --force FLAG, unless you're absolutely sure you know what you are doing

git push "[remote]" --tags
tags are not automatically pushed when you push a branch or use the --all flag. The --tags flag sends all of local tags to the remote repository.

Git Status

git status
displays the current state of you working directory and staging area

git status -s or git status --short
displays the current state of your working directory and staging area in a short format

git status -b or git status --branch
displays the current branch and tracking information in short format

git status -v or git status --verbose

displays the names of files that have been changed, and shows the textual changes that are staged to be commited (i.e., like the output of 'git diff --cached'). If -v flag is specified twice, it also displays the changes in the working tree that have not been staged (i.e., like the output of 'git diff').

Git Log

git log

displays a commit unique hash number, the develop who made commit, the date the commit occurred, and information about the commit (a brief summary).

git log --author="[author name]"

searches for commits by a particular '[author name]'. If found, returns commits by '[author name]'; otherwise, nothing.

git log --grep="[a pattern]"

searches for '[a pattern]' and displays only commits that have a message that matches '[a pattern]'.

git log --decorate

adds names of branches or tags of commits shown.

git log --graph

displays a drawn text based graph of the commits on the left side of commit messages.

git log -"[number]"

displays a limit of commits by the limited '[number]'.

git log --no-merge

will not display merged commits (note: by default merge commit are listed).

git log --oneline

displays a condense version of each commit to a single line.

git log -p

displays the full diff for each commit.

git log --pretty=oneline

displays commit in one line

git log --"[since]" --"[until]"
displays commits that occur between '[since]' and '[until]'. Arguments can be a commit ID, branch name, HEAD, or any other kind of revision reference (note: the --since and --until flags are synonymous with --after and --before, respectively).

git log --stat
displays which files were altered and the relative number of lines that were added or deleted from each of them.

Git Shortlog

git shortlog
displays each author name, followed by number of commits, and list of commits.

git shortlog -s
displays number of commits for each author.

git shortlog -"[number]"
displays the limit of the last '[number]' of commits for each author

Git Pull

git pull
updates your local working branch with commits from the remote, and update all remote tracking branches (note: git pull is shorthand for git fetch, followed by git merge [Fetch_HEAD]).

git pull --rebase
updates your local working branch with commits from the remote, but rewrite history so any local commits occur after all new commit coming from the remote, avoiding a merge commit (note: git pull --rebase is shorthand for git fetch, followed by git rebase [Fetch_HEAD]).

Git Rebase

git rebase
the process of moving or combining a sequence of commits to a new base commit. Rebasing changes the base of your branch from one commit to another making it appear as if you had created your branch from a different commit (note: It is

important to understand that even though the branch looks the same, it is composed of entirely new commits).

git rebase -i "[branch]"
interactively rebases current branch onto '[branch]'. Launches default editor to enter commands for how each commit will transferred to the '[branch]'.

Git Reflog

git reflog or git reflog show HEAD
records updates made to the tip of the branch. It allows the return to commits even to the ones that are not referrenced by any branch or tag. After rewriting the history, the reflog includes information about the previous state of branches and make it possible to go back to that state, if needed.

git reflog or git reflog show HEAD
brings up a log of all previous commits in the past.
- then use
git reset --hard <HEAD@{whichever number is next to the commit that is desire to go back to>
i.e. git reset --hard HEAD@{2} commit: "some message"

Git Reset

git reset
resets staging area to match most recent commit, but leave the working directory/folder unchanged.

git reset --hard HEAD~1
resets staging area to go back to last commit (Undo to last commit to GitHub)

git reset --hard HEAD~2
resets staging area to go back to last 2 commits

git reset --hard HEAD~3
resets staging area to go back to last 3 commits

git reset "[file]"
removes '[file]' from the staging area, but leaves the working directory unchanged (note: This unstages a file without overwriting any changes).

To remove unpublished commits (these are recent commits existing only on the local repository -- they have not been pushed to origin)

Run the command below:

```
git reset --hard <commit sha>
```

or

```
git log --oneline
git checkout <desired commit sha>
git add .
git commit -m "Reverting to <desire commit sha>"
```

or

```
git revert <desired commit sha>
git revert <branch>~2
```

Git Switch

`git switch "[branch name]"`
switches to the specified '[branch name]', if it exist.

`git switch -c "[branch name]"`
creates '[branch name]' and switches to the specified '[branch name]'.

Git Branch

`git branch or git branch --list`
lists all the branches in the repository.

`git branch -a`
lists all remote branches.

`git branch "[branch name]"`
creates a new branch with the name '[branch name]'.

`git branch -d "[branch name]"`
deletes '[branch name]'. If there are unmerged changes, Git does not allow you to delete it.

`git branch -D "[branch name]"`
forces delete '[branch name]', even if there are un merged changes. When you are sure to delete '[branch name]' permanently, execute this command.

git branch -m "[branch name]"
moves or renames the current branch to '[branch name]'.

git branch -r
displays the remote branches

Git Checkout

git checkout "[branch name]"
switches to the specified '[branch name]'.

git checkout -b "[branch name]"
creates a new branch named '[branch name]' base off the current branch and directly switches to '[branch name]'.

git checkout -b "[new branch]" "[existing branch]"
creates '[new branch]' based off '[existing branch]' and directly switches to '[new branch]'.

Git Clean

git clean -n
shows which files to be removed from the working directory.

git clean -f or git clean --force
executes the clean; that is, it removes the mentioned files shown in the 'clean -n'.

Git Diff

git diff
displays any uncommitted changes since the last commit.

git diff HEAD
displays the difference between working directory and last commit.

git diff --cached
displays the difference between staged changes and last commit.

git diff --color-words

git diff -highlight

git diff --line-numbers

Git Fetch

git fetch
retrieves commits, files, and references fom the remote repository into the local repository. Does not change your working state (note: It is harmless). It is more like checking to see if there are any changes available (note: It fetches but does not merge).

Git Merge

git merge "[named branch]"
executes combining '[named branch]' into the receiving current branch. It is used to combine multiple commits into one history.

Git Stash

git stash
takes uncommitted both staged and unstaged changes, saves them away for further use, and then returns them from your working copy.

git stash -a or git stash --all
adds changes to ignored files (note: git stash will not stash new files that have not yet been staged and ones that have been ignored).

git stash -u
includes to stash untracked files (note: git stash will not stash new files that have not yet been staged and ones that have been ignored).

git stash apply
re-applies the changes and keep them in the stash.

git stash apply --index
stages file that was not stages to stash

git stash drop "[stash name]"
removes '[stash name]' from the stash

```
git stash pop
removes the changes from you stash and re-apply them to your working copy

git stash show -p | git apply -R
un-applies the most recent stash
```