Lesson 2: Using `git`

- To initialize a git repository: `git init`
- What happens when you initialize a repository? Why do you need to do it?
  When I initialize a repository, it creates the `.git` folder and its standard subfolders. Now it's ready for the first commit.
- Git related directories: Working directory -> Staging directory -> Repository.
  Staging directory work as an intermediate step before commit. Use `git add file1 file2` or just say `git add --all`.
- Committing process:

```
git add file1
git commit #An editor will pop up to make changes.
# Alternatively use git "git commit -m "Message" "
```

- How is the staging area different from the working directory and the repository? What value do you think it offers?
  Only the changes or files mentioned in the staging area are going to be included in the final commit. Suppose we have multiple files changed since last commit, and we wish to include only a few files for the next commit since they constitute a logical change. We then include only those subset of changed files in the staging area.
- Use `git reset file1` to remove file1 from the staging area.
- Commit message style guide: http://udacity.github.io/git-styleguide/. Write in a imperative tone and end without period, for e.g., "Add PDF of file1" instead of "Added PDF of file1".
- Use `git status` intermittently to see which files has been committed, which are untracked etc.
- Using `git diff`:
  - `git diff`: compares working directory and staging area
  - `git diff --staged`: compared latest commit and staging area
- `git reset --hard`: Discards changes in working directory or the staging area
- Staging area always contains the last commit until a git add command is issued to move files from working directory to staging area.
- **Concept of branches**
  - master branch: default main branch in `git`
  - detached HEAD state: HEAD is the pointer of the commit in the working directory. When one issues `git checkout <commit_id>` to checkout a particular commit, then HEAD has no branch to attached to it, and throws this warning. Remedy is to create a temporary branch from the commit_id and delete it after use.

```
$ git checkout -b test-branch 56a4e5c08

...do your thing...

$ git checkout master
$ git branch -d test-branch
```

  - `git branch`: shows all the branches and with * on the current branch.
  - `git branch <branch_n>`: create a branch with name branch_n.
  - `git checkout <branch_n>`: checkout branch_n.

- What are some situations when branches would be helpful in keeping your history organized? How would branches help?
  - Experimental idea
  - A very new local To achieve this create a new branch and make it working. Once it start working, one can merge this into the main branch
- Showing git graph for some branches: `git log --graph --oneline branch1 branch2 branch3` for branch1-3.
- Idea of reachability: when one issues `git log` on one branch, it shows until its parents (look it as a directed graph). That's why some commits are not reachable by git log on some branches, and detached HEAD commits are not reachable by any branch.
- On a detached HEAD state issue `git checkout -b branchk`, where branchk is the new branch name. This command is equivalent to `git branch branchk` and `git chekout branchk`. Without issuing this command, all the changes made in this commit will be untrackable when move to any other branch.
- How do the diagrams help you visualize the branch structure?
  It shows at which commits diversion to various branches happened, merges, current branch etc.
- **Merging**
  - `git` intelligently merges two branches. When a branch is merged into master. The older branch is deleted, and all the commits of the deleted can be still accessed through master branch.
  - Merging creats a new commit, and the commit will have multiple parents.
  - Commands:

```
# Move to the master branch (or the branch to which to be merged)
git merge master coins
# If the merge is successful
git branch -d coins
# Abort Merging
git merge --abort
# Compare a commit to its parent.
git show 656b02e
```

- Merging conflict, for instance "CONFLICT (content): Merge conflict in game.js": It means that the same lines of code was changed by both the to-be-merged branches. So one has to look manually into the code and change. To solve this, open the file under conflict and seacrh for "<<<", and it will show the conflicting parts in the merged code. Then run the following lines of code to finish the merge.

```
git status #Should show the corrected conflict file as "unmerged paths, both modified game.js"
git add game.js #Add the conflict resolved file; Now it should show -
#"All conflicts resolved; changes made to be committed."
git commit
```

- Reflections: What is the result of merging two branches together? Why do we represent it in the diagram the way we do?
  Merging combines the changes in two branches in a smart way (to be clear).
- Reflections: What are the pros and cons of Git's automatic merging vs. always doing merges manually?
  Git's automatic merging makes it easy to merge non-conflicting areas of two branches. Manual merging will give more control, but of course with the tedious task of digging up the control. In some cases like *conflict detected* while automatic git merging has no choice other than using manual merging.