Model 2 Using Git Locally

1. Advanced Git Interaction

(1) Skipping the staging area

```
git commit -a
```

Function: directly stages any changes to *tracked* files and commits them in one step

- **Difference with *git add/commit***: does not work on new files which are untracked
- Can be combined with *-m* (write short *commit* messages only)
- No changes are allowed after *commit -a*, so make sure all changes you want are included in the *commit
- **HEAD** alias: represent the current checked-out snapshot of your project

(2) Getting more information about our changes

```
git log -p
```

Function: obtains the actual changes of lines in all previous *commits* (function equals to *diff -u* for all pairs of files before and after each *commit*)

```
git show <commit_id>
```

Function: shows the changes associated with a certain *commit* (with a *commit ID*)

- By providing the initial 4-6 characters, **Git** can guess what *commit ID* we are retrieving

```
git log --stat
```

Function: shows the details of what are modified (i.e., how many files are changed, with how many insertions and deletions)

```
git diff
```

Function: shows the differences between the modified but unstaged files (shown as +++ b) and the committed files (shown as --- a)

```
git add <file>(if all files selected, <file> can be omitted) -p
```

Function: shows the differences between the files that are being added to the staging area and the committed files, and generates a request of whether staging the files

```
git diff --staged
```

Function: shows the differences between the staged files and the committed files

(3) Deleting and Renaming Files

```
git rm <file>
```

Function: deletes the from the directory and stages this deletion action for the next *commit*

- The delete action is processed (by using *ls -l* you can see there is no such file in the directory) but has to be permanently acted by *commit*

```
git mv <filename_1> <filename_2>
```

Function: renames the file from *filename_1* to *filename_2* and stages this deletion action for the next *commit*

```
echo .DS_STORE > .gitignore
git add .gitignore
git commit -m 'Added a gitignore file to ignore .DS_STORE files'
```

Function: creates a *gitignore* file to ignore some files that are automatically generated to increase system noise

2. Undoing Things

(1) Undoing things before committing

```
git checkout <file>
```

Function: restores the unstaged file to the latest storage snapshot (either *committed* or *staged*)

```
git reset HEAD <file>
```

Function: unstages the modified file from the staging area

```
git checkout <file> -p
git reset HEAD <file> -p
```

Function: Perform the previous commands for files in a one-by-one manner

(2) Amending Commits

```
touch <file>
```

Function: creates a new file

```
git commit --amend
```

Function: Amends the previous *commit* and updates the *commit* using that from this round

- All the changes after previous *commit* are also committed in the updated *commit*
- The *commit message* can be changed in the amended *commit*, thus can be run even without changing any files
- The previous *commit* disappears from the *log* data, thus **shall not be used on remote or public repositories**

(3) Rollbacks

```
git revert HEAD
```

Function: reverts the repository to the version before the last *commit* version and creates a new *commit*

- Conducts a reverse action of the previous commit (i.e., for added lines, delete them; for new files, remove them, etc.)
- Automatically generates two lines of *commit message*:
  - *Revert "previous commit message"*
  - *This reverts commit .*
- Generally, we will add an explanation of why we are doing the rollback

```
git log -p -{number}
```

Function: obtains the actual changes of lines in the last {number} *commits*

(4) Identifying a commit

- ***Commit ID***: strings that appear after the word *commit* in the *log messages*
  - A *hash* calculated using an algorithm *"SHA1"* (part of cryptographic hash functions)
  - Used to guarantee the consistency of our repositories
  - Changes every time we amend a *commit* (the reason why not using *--amend* on public repositories)
  - Two *commit IDs* can **hardly** collide (happen to be the same) on purpose

```
git revert <commit ID>
```

Function: reverts the action conducted in the and creates a new *commit*

3. Branching and Merging

(1) Branch

- ***Branch***: a pointer to a particular commit
  - **Default branch**: *master* branch, git creates when a new repository is initialized
  - **Separate branch**:

- Use: want to try something new/develop a new feature/fix something without interfering with the main working state
- Action: can be merged to the *master* branch, or discarded without negative effect

(2) Creating new branches

```
git branch xxx
```

Function: a versatile command to list, create, delete and manipulate branches

```
git branch
```

Function: lists all branches in the repository

- the current *branch* is indicated with an asterisk * and in a green color

```
git branch <branch-name>
```

Function: creates a new *branch* with

```
git checkout <branch-name>
```

Function: check out the branch including both files and *git history* to whatever the head is pointing at

- When switching to a different *branch*, *git* changes files in the working directory and the *commit history*

```
git checkout -b <branch-name>
```

Function: creates a new *branch* with and immediately switch to that *branch*

- By checking the *commits* using *git log*, we can check the status of different branches

(3) Working with branches

```
git branch -d <branch-name>
```

Function: deletes the branch

```
git branch -D <branch-name>
```

Function: deletes the branch even if it has unmerged changes

(4) Merging

- **Merging**: Git uses it for combining branched data and history together

```
git checkout <branch1>
git merge <branch2>
```

Function: *merges* into (Make sure we are at by calling *git checkout*)

- ***fast-forward* merge**: occurs when all the *commits* in the checked-out branch are also in the branch that's being merged
  - **Action**: updates the older *commits* to the newest *commit*, no actual merging
- ***three-way* merge**: occurs when branches have diverged, e.g., when both *master* and extra *branches* have new *commits*
  - **Action**: ties all branch histories together with a new *commit* and merges the snapshots at the branch tips with the *commit* before the divergence
    - If the changes were made in different files/different parts of the same file -> takes both changes and puts them together
    - If the changes were made on the same part of the same file -> causes a ***!!!merge conflict***

(5) Merging conflicts

Merge conflict: happens when two different changes are made on the same part of the same file in the checked-out *branch* and the merged *branch*

- When executing a *merge* that causes a *merge conflict*, *Git* adds information in the file containing the *merge conflict*
- We can choose an option from "*accept the current (**checked-out branch**) change*", "*accept the incoming (**merged branch**) change*", "*accept both changes*", and "*compare changes*" (no action; pops up a new window comparing both changes)

```
git commit
```

Function: commits the *merge* after resolving the *merge conflict*

- Shows the information "*Merge branch ''*" and the conflict information
- We can add a line describing the option we have chosen during resolving the issue

```
git log --graph --oneline
```

Function: shows the *commit history* in a graph format with only one line for each *commit*

```
git merge --abort
```

Function: if there's a merge conflict, aborts the processing *merge* and resets the files in the working tree back to the previous *commit* before the *merge*