1. `diff <name of code 1> <name of code 2>`: show the difference between two codes.
    a. 5c5,6 --> line 5 at code 1 changed with line 5,6 at code 2 (c means changed)
    b. 11a13,15 --> line 11,13,15 added.

2. `diff -u <name of code1> <name of code 2>`: to show better the differences, we can use the unified version.
    a. In output: "-" sign means lines which are deleted, and "+" sign means lines which are added.

3. **wdiff, meld, KDiff3, and vimdiff** are other tools.

4. `diff -u <name of code1> <name of code 2> > <change.diff>`: saving the `diff` result to another file.

5. `patch <output file name> < <filename.diff>`: `patch` command takes a file generated by `diff` and applies changes to the original file.

6. Files are usually organized in **repositories**, containing separate software projects or just group all related code.

7. **commit** is a collection of edits which has been submitted to the **version control system (VCS)** for safe keeping.

8. `git config --global user.email "<email_address>" user.name "<your name>"`: To show git who we are, we should use `git config` command and specify our user name and user email
    a. `--global`: means that we use this name and email for all repositories.

9. `git init`: start a repository from the scratch ----- `git clone <URL address>`: start a repository that already exists somewhere else.
    a. After creating a repository with `git init` **-->** we can use `ls -la` to find it in the directory.
    b. Also `ls -l .git/` command help to see the files in the created **Git repository (Git directory**) **-->** stores the changes and the change history.
    c. The area outside the git directory is the **working tree**.
    d. The working tree is the current version of your project **(workbench).**

10. `git add <file name>`: to make git track our files.
    a. we add our file to the **staging area** (index): means a file maintained by Git that contains all information about what files and changes are going to go into your next commit.
    b. `git add -p`: show the difference before adding to staging area.
    c. `git add *`: This command will end up adding any change done in working tree to the staging area.

11. `git status`: getting some information.
    a. changes to be committed: .... (they are in the staging area)

12. `git commit`: to get it committed into the `.git` directory **-->** it will open a text editor to write a commit message.
    a. `git commit -m '<your message>':` adding commit message in one line.
    b. `git commit -a`: a shortcut to stage any changes to tracked files and commit them in one step. By using this we skip staging area, means we cannot add any other changes before creating the commit.
    c. `git commit a- -m " <commit message>":` all of the steps can be done in one command. This shortcut is good for small changes.
    d. `git commit --amend`: allow us to modify and add changes to the most recent commit. Avoid amending commits that have already been made public because it will overwrite the previous commit.

13. Any Git project will consist of three sections. The **Git directory**, the **working tree**, and the **staging area**.
    a. The **Git directory** contains the history of all the files and changes.
    b. The **working tree** contains the current state of the project, including any changes that we have made.
    c. The **staging area** contains the changes that have been marked to be included in the next commit.
    d. Each track file can be in one the three main states: **modified, staged, or committed**.

14. `git config -l`: show git repository information.

15. When we create a file in a repository (using `git init` or `git clone` [copy]), the git does not track the changes into file until we add it (`git add`).

16. The good git **commit message** should be like this:
    a. 50 characters tell the summary.
    b. 72 characters tell the detailed.

17. **Git Steps:** `git init/git clone` --> `git config` --> `git add` --> `git commit`

18. `git log`: shows the history information about all commits.
    a. `git log -p`: show the patch that was created. To see the changes details in each commit.
    b. `git log --graph`: seeing the commits as a graph.
    c. `git log -- oneline`: see one line per commit.
    d. `git log --stat`: show some stats about the changes in the commit.

19. `git show <git commit id>:` show the changes for specific commit.

20. `git diff`: it helps us to track the difference before committing.

    a. `git diff <name of file>`: If we want to see specific file diff rather than all file in repository.

21. `git rm <filename>`: remove the file from the git repository.
    a. After that we need to commit it.

22. `git mv`: to move files between directories.

23. **git cheat sheet**: https://training.github.com/downloads/github-git-cheat-sheet.pdf

24. `git checkout <filename>`: change a file back to its earlier committed state (it reverts changes to modified files before they are staged)
    a. `git checkout`: use to check out the latest snapshot for both files and for branches.
    b. `git checkout <name of branch>`: switch to the specified branch.
    c. `git checkout -b < name of branch>`: creating a new branch and switch to it.

25. `git reset`: remove from the staging area.
    a. git reset -p: used for specific changes.

26. `git revert`: create a new commit, opposite of the changes in the bad commit.
    a. `git revert <commit ID>`: This can be used for rollback a commit that wasn't most recent one.

27. Each git commit has an **ID** which is created by SHA1 algorithm using information related to the commit. It also brings consistency.

28. **Branch**: a pointer to a particular commit. It represents an independent line of development in a project.
    a. The default branch that git creates for you when a new repository is initialized is called **MASTER**. The master branch is commonly used to represent the known good state of a project. So, it is better to commit the last code in there and use another branch for testing.

29. `git branch`: show all the branches in your repository, "*" the asterisk show which branch we are.
    a. `git branch <branch name>`: creating a new branch.
    b. `git branch -d <branch name>`: delete the branch.
    c. This demonstrates that when we switch branches in git, the working directory and commit history will be changed to reflect the snapshot of our project in that branch. When we check out a new branch and commit on it, those changes will be added to the history of that branch.
    d. `git branch -r`: looking at the remote branches that our Git repo is currently tracking.

   e. To modify the branch contents: we pull any new changes to our local branch, then merge them with our changes and push our changes to the repo

30. **Merging**: The term that Git uses for combining branched data and history together.
    a. `git merge <branch name>`: merge the branch to the master branch.
    b. `git merge --abort`: If there are merge conflicts (meaning files are incompatible), --abort can be used to abort the merge action.

31. If you do not want to set up a Git sever yourself and host your repositories, you can use online service like **GitHub**.
    a. **GitHub** is a web-based Git repository hosting service. Other examples are **GitLab** or **BitBucket**.

32. `git clone <URL address>`: copy the repository in your local computer.
    a. `git clone <URL address> <directory name>`: copy the repo in specified directory.

33. `git push`: send changes to that remote repository (update the remote repository). [commits from local to remote]

34. `git pull`: get changes from the remote repository. [fetch the newest updates from a remote repo]

35. To avoid having enter the password each time we push or pull. We can do these ways:
    i. create an SSH key pair and store the public key in our profile so that GitHub recognizes our computer.
    ii. Use credential helper, which caches a password for a limited time.
    iii. `git config --global` credential.helper cache

36. For synchronization of all codes, we should consider these stages when we are doing collaborative project.
    a. modify --> stage --> commit |local changes in local repo| --> fetch |new changes from remote repo |--> merge | if necessary| --> push)

37. Git set up the remote repo with the default origin name when we call a git clone.
    a. `git remote -v`: in the directory of the repo show the configuration (the fetch and push URL can be different in some cases) [verify that you have already setup a remote for the upstream repository, and an origin].
    b. `git remote show <origin or other name>`: a complete information about the remote origin
    c. `git remote update`: get the contents of a remote branch w/o auto merging.

38. Git does not keep remote and local branches in sync automatically.
    a. `git fetch`: This command copies the commits done in the remote repository to the remote branches.

        i. the difference between `git fetch` and `git pull`: `git fetch` only fetches remote updates but `git pull` fetches + merges
        ii. the difference between `git fetch` and `git remote update`: `git fetch` download but the other one show
    b. `git merge origin/master`: merge the changes of the master branch of the remote repo into our local branch.

39. `git push -u origin <name>`: adding branch to a remote repository, having branch reduces the problem in merging.

40. `git rebase < followed by the branch name want to set as the new base>`: change the base commit that's used for our branch
    a. keeping history linear helps with debugging especially when we are trying to identify which commit first introduced a problem in project.

41. **Forking**: a way of creating a copy of the given repo so that it belongs to our user. our user will be able to push changes to the forked copy, even when we cannot push changes to the other repo -- there is a button in **GitHub** for forking the selected repo in our repo.

42. **pull request**: after changing and modifying the selected project, we can pull request to notify the developers of that project to apply our changes.