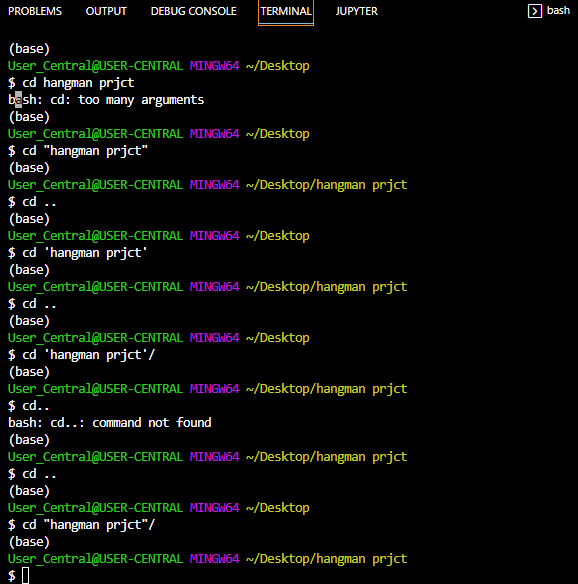
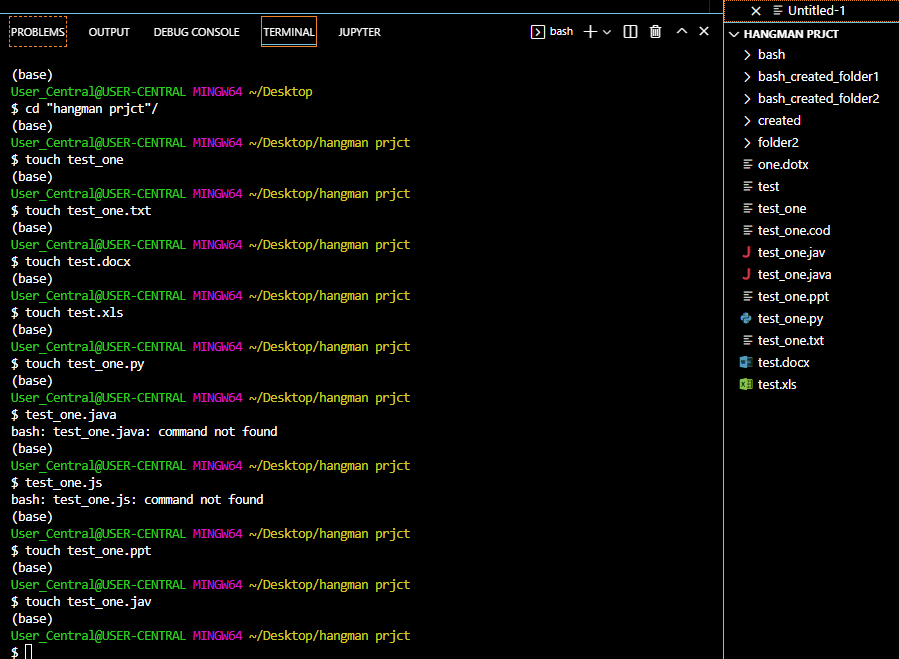
*The* **cd** *command*



Wrong way to cd (change directory ) to a folder that has two names not joined with an underscore i.e folder name is not one word.

Correct way to label and cd to a folder i.e in quotes like a string; along as the folder name is not one word



Git

git is currently the most utilised distributed **version-control system** (VCS) and the de facto standard for code collaboration globally.

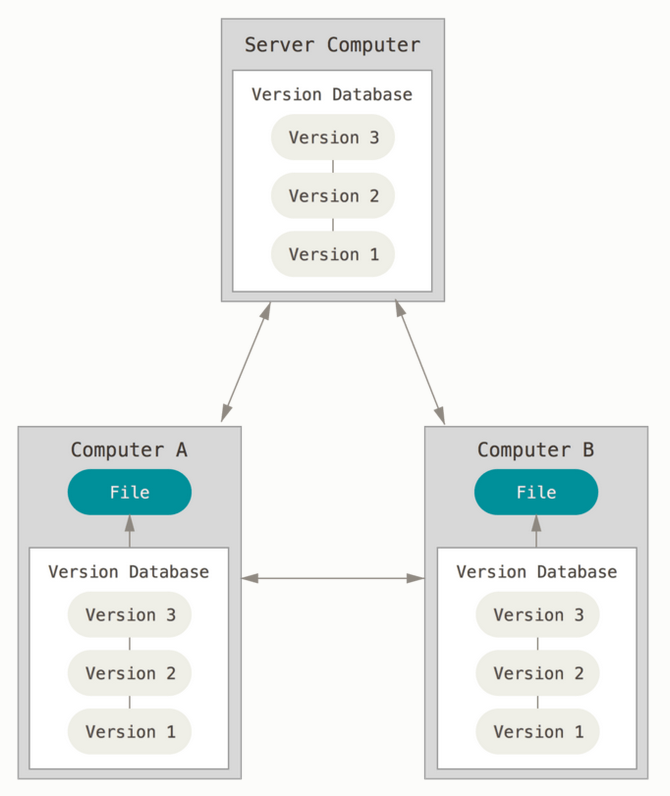
As a program, git can be run in the command line, which we are slightly familiar with. If you are on a Windows machine, use gitbash; otherwise, use the default terminal.

Version-Control System

First, what is a VCS?

As the name suggests, a VCS refers to a software utilised for tracking changes to code and maintaining the records of saved versions.

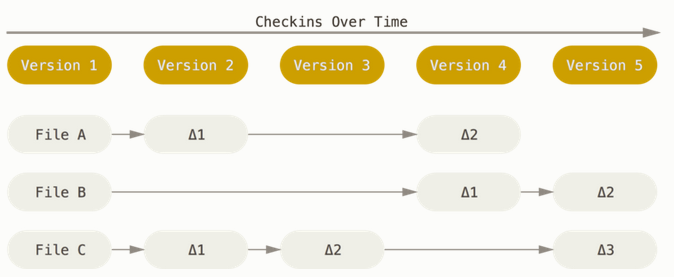
A **VCS** records changes to file(s) over time, thereby providing constant access to previous code **versions.**



Git is considered to be **distributed** because each node (client) **mirrors the full repository and its history.**

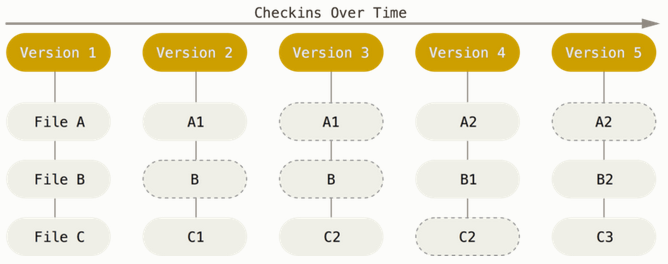
Git Features

Other than git, other VCSs, which are comparatively unpopular currently, **store information about file changes** (see below).



Thus, these VCSs only consider a file if a change has been made. In other words, the new state does not consider files that have not changed.

Conversely, git records 'snapshots' of the whole code, known as commits. Think of a commit as a picture of all the files at a certain point. With that picture, git can always revert to that state.



Repositories

When you tell git to start keeping track of the files in a folder (or directory), you move the commits to a git **repository**, in which git stores the snapshots of the files in the working directory.

As mentioned previously, git is a distributed VCS; however, that does not mean that you change the state of the central server whenever you work on your repository. When you work on a repository, your changes are saved locally and will not be reflected on the central server until you push them.

Note that git only adds data. This implies that the operation of removing a file could be considered as 'add file deletion'. Thus, you will not lose data if you commit your changes frequently. You **ESPECIALLY** will not lose your data if you push your changes to a central server (as you probably can tell, we are gradually approaching GitHub).

State of any File in git

The files in a repository can be in one of three stages:

* Modified: You changed the file but are yet to commit the changes to your database.
* Staged: You have marked a modified file in its current version as being ready to go into your next commit (snapshot).
* Committed: The data are safely stored in your local database.

For clarity, here is an example of the conventional flow of a file in a repository:

1. The file is created or modified. Thereafter, git compares the changes in the current directory with those in the last snapshot and notices some changes.
   * The revised file is now labelled as **modified** because it changed with respect to its last snapshot.
2. When you have completed a session of revisions on the file, you can mark the file as being ready for the snapshot, thereby positioning it in the **staged** state.
3. git takes the snapshot of the file and stores it in the repository. The file is now **committed.**
   * The next time you change a file in the working directory, it will enter the modified state, and the process repeats once more.

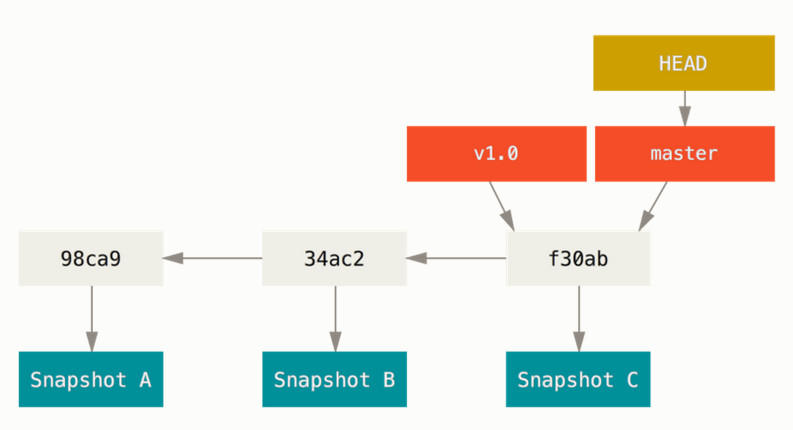
Although 'snapshots' are employed in the above explanations for clarity, the technical term is **commits**.

Branches

One of the powerful features of git is **branches**.

Branches are **movable pointers to commits**. Think of them as separate paths in code development, which you can later merge.

* By default, git creates a branch called main (formerly master) after running the git init command. **You should always keep it as your main branch.**
* HEAD (which we have encountered) is a pointer to the current location in the commit history.



**Using branches, one can**

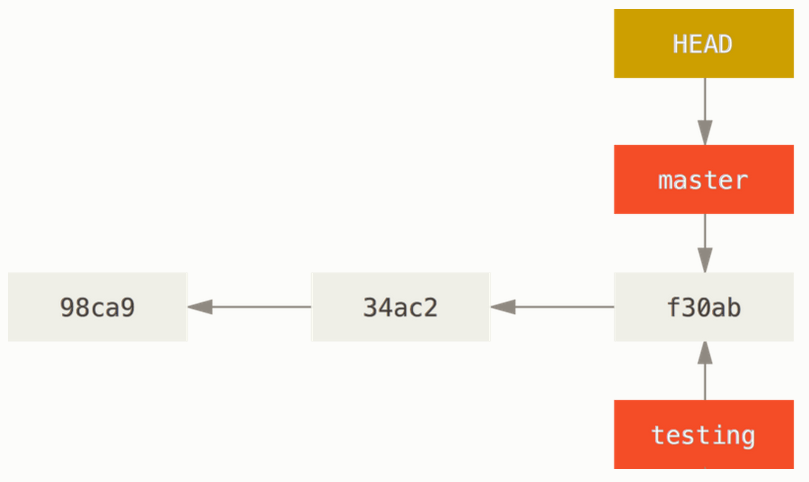
* work on new features separate from other developers.
* ensure that the entire process and workflow are structured and easy to follow
* test out experimental/work-in-progress (WIP) code without altering the **master branch**

Use branches **ALL THE TIME.**

Working with Branches

git branch NAME\_OF\_BRANCH is the basic command responsible for creating branches.

Below, we illustrate what happens after we run the git branch testing command.



Few things to note:

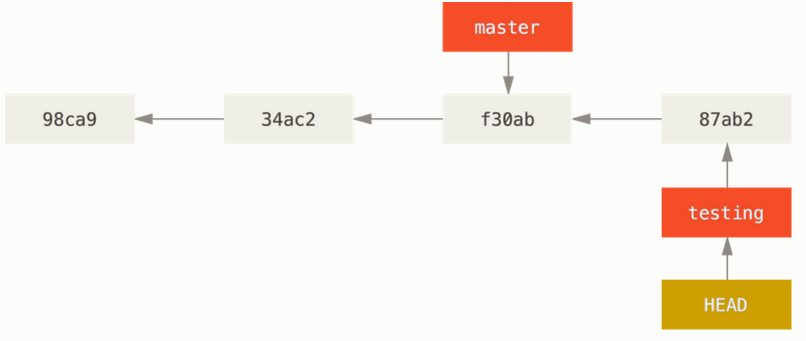
* **We are still on the master branch, as indicated by ~HEAD.**
* The new branch is merely a pointer to the last commit.

To switch to the new branch, we run the git checkout command:

git checkout testing

Tip: git checkout -b NAME\_OF\_BRANCH creates a new branch and checks it out (i.e. switches to the new branch) automatically.

Now that we are on the testing branch (HEAD points to it), we can perform the usual operations, including git add and git commit, and achieve results, as shown in the figure below:

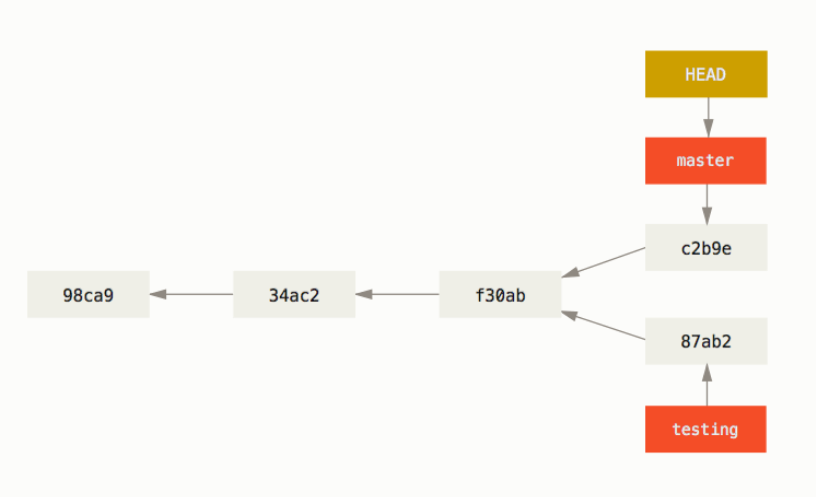


We can switch back to the master by simply running the git checkout master command.

Things to note:

* **Your local changes will revert to how they were on the master.**
* **This does not imply that your changes are lost. They are simply committed on another branch.**

Now, we commit on the master branch as well, which leaves us with the following (divergent) branch structure:



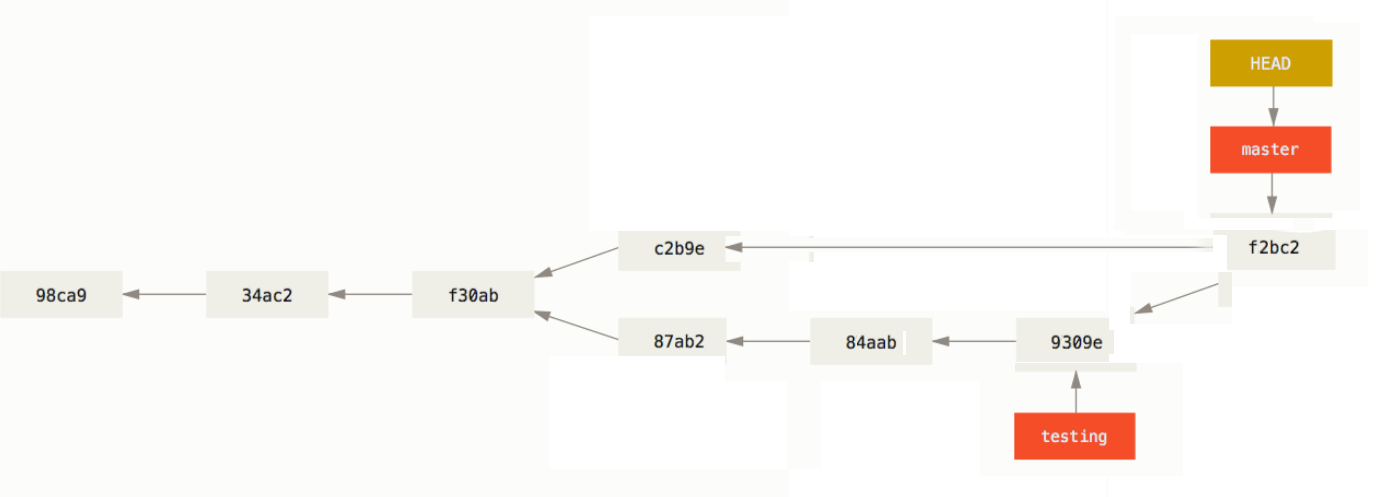
Tip

* **Pull all changes from the remote repository before creating a branch with new features.** This will minimise the risk of merge conflicts.

Merging

When you have completed your work on a branch, you can apply the changes to the main branch by merging it.

From the example above, let us assume that you have completed your work on the 'testing' branch. You can implement the changes from 'testing' into 'master' by merging.



There are three methods for merging a branch into another:

* merge commit
* squash and merge
* rebase and merge

In most cases, the default 'merge commit' will suffice. To obtain an in-depth understanding of the differences between these merging methods, read the following StackOverflow [thread](https://stackoverflow.com/questions/2427238/what-is-the-difference-between-merge-squash-and-rebase).

Exercise

Admittedly, that was a lot to process. Now, we put into practice all that we have learnt by creating a local repository, adding files to it, and finally committing those files. Thereafter, we will experiment with branches to improve your understanding of their working mechanisms.

Please carry out the following tasks and observe the changes in your local machine.

1. Create a new directory on your Desktop named 'AiCore\_git'.
   * Navigate to your Desktop from the terminal using the cd command.
   * Create the directory using the mkdir command.
2. Change your working directory to 'AiCore\_git'.
   * Once more, use the cd command.
3. Run git init to create a repository.
   * This will create a hidden directory that contains all the information regarding your commits.
4. List the files contained in 'AiCore\_git'.
   * Use the ls -a command to display all the files, including the hidden ones.
   * Notice that a directory named .git has been created.
5. Create two different files, e.g. 'test\_1.txt' and "test\_2.txt'.
   * Use the echo or the touch command for this task.
6. Check the status of the directory.
   * Run git status.
   * Read the message, and attempt to understand the state of your files.
7. Move the files to the **staged** state.
   * Use the git add command, followed by the name of a file to be staged.
   * Alternatively, you can stage all the files using the git add . command.
8. Check the status of the directory again.
   * Rerun git status.
   * What differences do you see with respect to the output of the previous git status command?
9. Take a snapshot of your new files so that git remembers them. In other words, make a commit.
   * Use the git commit command to commit all the files in the staged state.
   * Remember to add a commit message. Add the -m flag to the command, followed by your desired message in quotes.
     + For example, git commit -m "First commit".
10. Once more, check the status of your directory, and observe the differences.

Using branches

Here, we experiment with branches.

1. In 'AiCore\_git', create a new branch named 'testing'.
   * Use git checkout -b testing.
   * Here is a breakdown of the command syntax:
     + git checkout: switches to a different branch
     + -b: creates a new branch
     + testing: refers to the name of the new branch
   * Basically, we are creating a new branch called 'testing' and switching to it immediately.
2. Check the active branches in your directory.
   * Use git branch and see the output.
3. Create a new file named 'test\_3.txt'.
4. Stage and commit 'test\_3.txt'.
5. Switch to the main branch.
   * Use git checkout with the name of your main branch (Conventionally, it is either main or master).
6. List all the files contained in the directory.
   * If done correctly, 'test\_3.txt' should be out of sight.
   * However, do not be alarmed; 'test\_3.txt' is stored on the 'testing' branch. Note that none of the changes made on the testing branch were applied on the main branch. This is why it appears to have vanished.
7. Merge 'testing' into the main branch.
   * Use git merge testing.
8. Once more, list the files contained in this directory.
   * Great, we can see that 'test\_3.txt' is now in the main branch.

As you can now tell, using branches is a great way to not compromise your main code base.

Reverting Changes

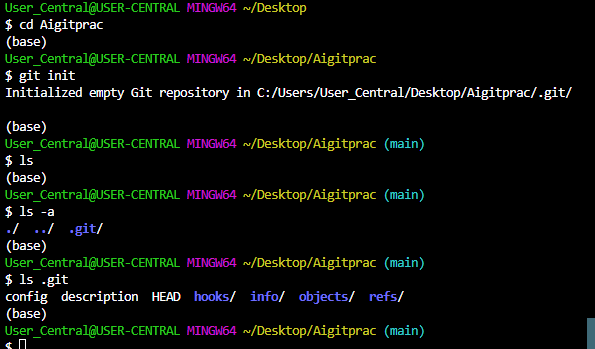
If you accidentally add too many files and commit in a hurry, you can easily revert to the 'pre-changes' state.

For that, we can use the git reset command.

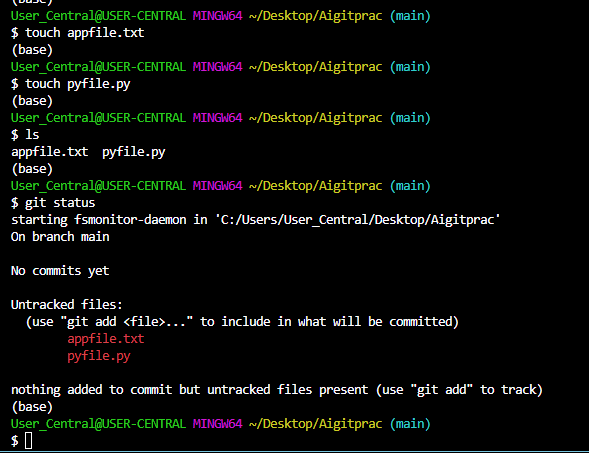
* git reset HEAD~ (the HEAD is actually written, it is not a placeholder here): reverts the last git commit and unstages (reverts git add) the files (you have to run git add to stage them again). Other than these, **no change will be made to the files. Therefore, you may rest assured that the files WILL NOT be deleted.**
* git reset [FILE]: reverts git add; if FILE is specified, it unstages the file; without any arguments, it unstages everything.

Resources

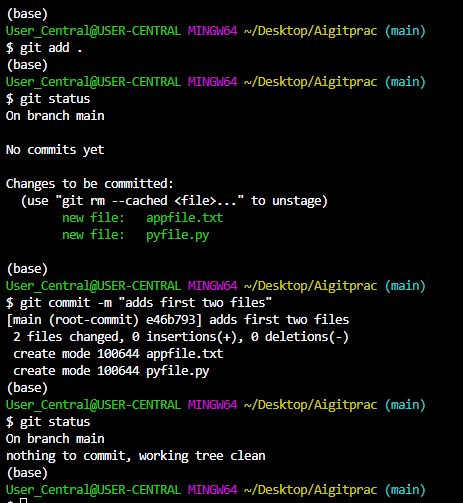
* [Pro Git Book](https://git-scm.com/book/en/v2) is one of the best resources on git (it is also a reference for some of the information provided herein).



This directory repo stores all the info about the entire history of changes to your code. It also stores the current state of your code and the branch you are on



Files are in Modified state waiting to be staged (preparing files to be included in next commit) once we are happy with the changes



commited files using [*git commit –m “meassge”*] command.

Staged files using [*git add .* ] command.