Version control, also referred to as source code management (SCM), is a software practice that extends beyond code and can handle assets as well. Its primary purpose is to track and document changes made to code by recording who made the changes and when. They allow you to go back to an old version of the code, or of just some files, and facilitate merging of conflicting modifications. Although the fundamental functionalities are generally consistent across various Version Control Systems (VCSs), the more advanced features can vary significantly. Additionally, the terminology used may differ between systems. One of the most commonly used VCSs is "git," which will be the focus of our discussion here. The crucial aspect of version control is its implementation, regardless of the method chosen, as long as it is effective. Version control has a history as old as computers themselves. In 1959, the US National Archives Records Service stored code on punched cards, achieving a data density of about 100MB per forklift pallet. Significant programs were archived, and any changes required creating an entirely new card deck. The emergence of UNIX in the 1970s marked the introduction of file-system-based version control, enabling storage of file modifications and user permission management. Git was created by (and named after) Linus Torvalds (of the Linux Operating System) in 2005, because the system they were using, bitkeeper, removed its free community edition. Git shares many of the useful features developed by earlier version-control systems. In particular:

• Moved/renamed/copied/deleted files retain version history.

• Commits are atomic.

• Sophisticated branching and merging system.

• Used a distributed storage system, where each developer has as much of the repository as wanted in a local copy and merges onto central server when ready.

Once you have installed git, you first want to set up some basic information. We noted that git stores the author of every change, and this means you have to provide your identity. After that you can create a repository. Now you can add files to the repo. You usually do this in two steps. First you add, or stage the change, which is get things ready, and then you commit. You can add multiple files, or parts of files, before carrying on. Git can show you a list of differences between two commits, or a list of differences between a given commit and the current state using the command “git diff”. Reversing changes in Git can be complex due to its distributed nature. When code has been shared with others, altering history directly can disrupt their states. Instead, "reverts" are used, creating new commits that undo previous changes to restore the original state. Both the original commit and the new revert commit are preserved in the log. Notably, if sensitive data like passwords or personal information is accidentally committed, using "git revert" will not eliminate it from the repository. If you are working on several things at once, you may find branches useful. These are versions of code that git keeps separate for you, so that changes to one branch do not affect another. Whenever you create a repository, a default “master” branch is created. The branch is based on the last commit (on whatever branch you are on when running the command). Note that if you have uncommitted changes when you run git branch, those changes will come with you, and can be committed. If you try and change branches when you have uncommitted changes, you may get an error, saying error: Your local changes to the following files would be overwritten by checkout:. You can either commit those changes, or consider using “git stash” to preserve them to use later. When you use branches to develop features, you usually eventually want to bring them back into the main version, when they’re ready to be shared with users, or are fully complete. This is a merge. Merging in Git can be aided by tools, but the process can grow intricate. While Git attempts to comprehend the code's structure, handling merges involving code modifications remains a complex challenge. For instance, altering the indentation of an entire code block might result in the block being removed and then added again during a merge, which can manifest as a merge conflict. This showcases the intricacies of reconciling changes in code, highlighting the complexities involved in automatic merge resolution. Git is a distributed, networked version control system, which is the core of its real power. You can link between a local repository and a remote one, on a server, or on GitHub, and git remembers that. You can clone code from a remote repository and git will remember the origin. To clone code, you need the URL of a remote server. When the copy of the code on the remote is updated, you will need to pull in those changes. This happens on a per-branch basis. Note that there is a related command, fetch, which just updates branch information and downloads changes, but doesn’t merge them into yours. If your local copy has also changed, you will have to deal with merging changes from other developers with your own. To upload your changes to the remote, you can push them. This happens on a per-branch basis. Note that there is a related command, fetch, which just updates branch information and downloads changes, but doesn’t merge them into yours. If your local copy has also changed, you will have to deal with merging changes from other developers with your own. To upload your changes to the remote, you can push them. Version numbers whenever you make a change, and to write this version number into your files. One very common, and very useful, versioning system is to use 3 numbers combined into a string like “1.1.4”.