Git for computational physicists

raul

December 9, 2022

Git's own webpage starts with:

> Git is a free and open source distributed version control system [...]

We will need to define two terms to understand this sentence:

- Version control system (VCS): A category of software tools that helps in recording changes made to files by keeping a track of modifications done in the code.
- Distributed: Meaning collaborative, in the sense that it can understand several copies of the project existing and contributing changes to each other. There are other types of VCS, but they are basically reduced versions of distributed ones and we will not cover them.

There are many VCS tools, but our choice is going to be git, the de-facto standard [1].

Use git correctly and your peace of mind will reach nirvana levels. Do not be fooled, though, as git's power is only surpassed by its dangerousness. Think of git as a chainsaw; hand it to an experienced lumberjack and you will be warm next winter, but let a drunken monkey take care of it and see what happens... Gits is infamous for its obscure syntax, with command names that transmit no information or are misleading[2] and lots of contextual behavior[3]. Many times the same action can be performed in several ways and other times the same command can be used to deal with completely orthogonal situations.
These situations may leave you wondering about the strange design choices of git. The key here is that there was (kinda) not a design choice involved. Git's command line interface (CLI) evolved organically over the years to accommodate new necessities[4] and most of the time this explains its oddities.
You are not forced, though, to use git's CLI to leverage git. There are several interfaces, textual and graphical, which use git under the hood, calling it for you when dealing with the typical workflows in a version-controlled project. Most IDEs have one (like VS code or emacs[5]), and there are also standalone ones (like Github Desktop).
In the following lessons[6] we will learn the basics of git, which will greatly improve your personal workflow and the way you collaborate with others when dealing with text-based files, such as latex papers, reports and software. Come with me to the depths of git hell (and actual concept that we will cover later) and let's have fun.

1 Why do you need a distributed version control system (VCS)?

Some benefits of a VCS:

1. Enables e cient collaboration (multiple people

[1]There is also subversion (svn), mercurial, ...

[2]`git cherry-pick` is an actual command.

[3]The command `git checkout` can do things like transport the entire project to a different point in time, delete a file, resurrect a file and more depending on the name we give it as next argument.

[4]Git was created by Linus Torvalds to version control the Linux kernel codebase.

[5]The one in emacs is called magit, and it is life-changing

[6]There are countless resources on git online, GitHub provides a good one, git-scm is also really good.

can work simultaneously on a single project).

2. Allows one developer to work on the same project from multiple computers.
3. Enables traceability of every small change.
4. Informs us about Who, When, What, Why changes have been made.
5. Each developer keeps and maintains a local copy, which are only merged after validation.
6. Allows to visit a snapshot of the project at any point in time.

Maybe these are a little abstract and you are not convinced, I will present to you a couple of nightmarish short stories that will make you cry out for a VCS:

1.1 The journal

Five collaborators are working on a draft for a new paper in LaTeX. Each of them is working on a di erent section, but often modify other ones (introduction, abstract...).

It has been three weeks since the last time anyone shared their version of the draft. Now lets switch our perspective to you, the Ph.D. student that has received five .tex files in the previous days in their mail, accompanied by a bunch of figures. To make matters worse, many of the figures are called "fig1.eps".

Naturally, each writer started their contributions at a di erent point in time, and thus from a di erent version of the .tex file. So you ended up with SIX versions of the same .tex file, and you are tasked with merging them all. Jumping to three months in the future, you now live in a psychiatric institution. On the good side, you no longer have to try to cosplay as git[7].

1.2 The bug

Last year, part of your research required you to develop a small post-processing software (about 3000 lines) that proved to be more convenient than you expected. You sent it to your advisor in an email because they had similar needs.

With time your advisor's needs for the code evolved and they, along other members of the group, patched it up adding new functionality and adaptations. Your own needs also evolved and you patched it in another way.

At this point there are several versions of the software lying around, all of them with similar, but not quite, functionalities. Several members of the group use some form of the post-processing software and many articles are being cooked that rely on it, some of them have even been published already. Some time ago you found out some surprising and novel results that you decide are worth publishing in some prestigious journal. After a painful process of collaboratively writing a paper (you are also not using versioning control for this article's latex source) the manuscript finally reaches the referees. One of them has had a lot of experience with the kind of post processing you use and notices something weird with one of your figures.

You religiously check your results and after a tedious process of software archaeology and intense testing you realize there is a critical bug in your software and pretty much all your surprising results are a consequence of it.

It has been at least a year since the software escaped your control when you emailed some version of it. You no longer have any recollection, let alone a proper log, of what changes occurred when. You do not even know if the bug happened before or after the last time you shared the code. As a matter of fact, the exact code used for some of the articles does not exist anymore, as it was overwritten during its evolution[8].

Once you find the problem, you can fix your personal version of the code by modifying just a couple of lines [9]. But... How much time will have to be spent finding out which articles will have to be retracted? How many hours will have to be spent on tracking

[7] Git is perfectly fine for Latex, but you could also use something like Overleaf to prevent this situation.

[8] God forbid some hard drive failed during this time and you lost everything.

[9] The vast majority of bugs I have encountered are fixed by replacing just one or two characters.

the children of this software making sure they did not inherit your bug? You do not know, because you now reside in a psychiatric institution.

I can tell you, though, what the situation would have been if you and your peers would have had used version control (such as git or svn):

1. You could have traveled to any point in the history of the software with a single **`git checkout`** call, allowing you to narrow the point in time (the commit in VCS terms) when the bug was introduced.

2. It would have taken you a single command, **`git blame`**, to find out the exact second the bogus line of the code was introduced and by whom, with a comment explaining the rationale of the change.

3. It would have taken you less than a minute to push a commit (terms/git commands you will come to know later) that fixes the bug in your code.

4. It would have taken others 3 seconds to incorporate your fix in their own children versions (yes, even when they are descendants of your original code) with a combination of pulling, merging and cherry-picking (more terms/git commands that we will go through).

5. Bonus: You use a platform like github, so your software lives in the cloud and is safe against any damage to your group's hardware.

Granted, the severity of the second situation could have been reduced, if not avoided entirely, if you followed a healthy software development workflow that includes things like unit testing, thorough documentation, proper code comments, etc. But chances are that if you are not using version control you are also neglecting many of the rest.

Now that you are convinced that you need VCS in your workflow we can move on.

2 Basic concepts

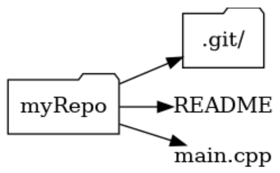
We will download a repository and work with it through examples. From now on, I will introduce new git commands by examples placed in blue boxes. Light gray boxes denote curiosities and/or technical details that are not that important.

2.1 Repository

A repository (or simply repo) is a collection of files accompanied by a database of changes. This database contains all the edits and historical versions (snapshots) of the project

In git, this database is stored in a folder called .git in the root directory of the project. If you remove this folder you would still have the project's files at the current point in time, but you would have lost all information about its history or about the location of any remote copy of the repo.

Before we move on, it is useful to show an explicit example of what a "project" might be. Say we have a simple project called "myRepo", composed by a README file and a single C++ source file:



A copy of the repository stored somewhere that is not the local copy is referred to as a remote. The default remote when you clone a repository is called "origin". Many remotes can exist in a repo, although most of the time origin will be enough. You can work with remotes by using the **`git remote`** command.
Try to run **`git remote show origin`** in your copy of the UAMMD repo.

Cloning a repository

To obtain the contents of a remote repository, you have to clone it. Lets go ahead and clone UAMMD

```
$ git clone https://github.co
↪   m/RaulPPelaez/UAMMD
```

This command will create the UAMMD directory, cd into it and inspect it.

Getting help from git

One you have some notion about a git command, you can obtain more information about it (such as the options it allows) using git help. Try to run the following in your terminal:

```
$ git help clone
```

We can also create a repository of our own from a project which is not yet version controlled. Lets start by creating a folder structure Lets start by creating a folder structure:

```
$ mkdir myRepo
$ cd myRepo
$ echo "This project is called myRepo"
↪  > README.md
```

Now we can use `git init`.

Create a new repository

Get into the root directory of the project (for instance, the myRepo you just created) and run:

```
$ git init
```

If you run `ls -a` you will see the .git folder was created.

For now, lets keep working on UAMMD.

2.2 How a repository stores snapshots

Each snapshot of the project is identified with a commit. Commits are named with an unique alphanumeric hash, for instance:

`d669805bfd9384017438d712ca3c55088c17aa30`

Commits contain information about a set of changes in addition to information including a timestamp, an author and a description.

Showing information about a commit

The git show command will give you information about a certain commit given its hash.
Lets inspect one small commit in UAMMD (the latest at the time of writing).
Get into the UAMMD repo you cloned before and run:

```
$ git show 917e1942328b8d9a5a⌟
↪  f4d0221c1a6c14fff8020f
```

The command **git help show** will tell you of the di erent ways of visualizing this information. Referring to the commit as simply 917e also works, as it is not an ambiguous (i.e. no other commit hash starts with that string). There are, however, other commits that start with just "917". See what happens if you try **git show** 917.

We can typically refer to a commit just by the first characters, since in most repositories that also constitutes an unique identifier (like "d669805"). Our tools will complain when we try to refer to a commit using an ambiguous hash (for instance if we refer to a commit with a short hash that is too short).

Internally, git does not store the totality of the project at every commit, rather it stores the first version of the project and then a list of changes that take it from one commit to the next.

We can thus represent the history of a repository using a list of connected nodes (representing commits):

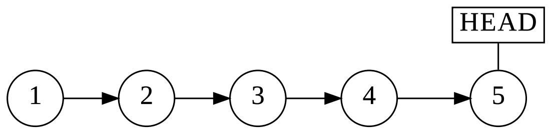


Figure 1: Each number represents the hash of a particular commit. Being 1 the first commit and 5 the current one (the HEAD).

A repo allows accessing a list with all the commits since its creation. We can use the **git log** command to navigate it.

Inspecting a repo's commit history

The **git log** command is used to navigate a repositorie's commit history.
Get into the UAMMD repo you cloned before and run:

```
$ git log
```

Maybe the default shows too much information. Play around with the help command now. For instance, try:

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

which shows only a short hash and the first line of the description for each commit. The view you get is equivalent to the representation in figure 1.

The HEAD commit is an alias for the current commit the repository is pointing to. You can use HEAD wherever a commit's hash would be valid (commands like "show", "checkout", etc). You can use HEAD to refer to commits relative to the current one by using $\sim$, for instance HEAD$\sim$ 1 refers to the commit just before the current one.
Notice that **git log** marks the current commit as HEAD.
With these tools you can glance at the history of a repo. You have the ability to know when changes happened, what the changes where and who did them. However, the repo is still sitting at the latest commit, the one you got when you ran **git clone**.

Besides showing the information for a commit, with git we are capable of visiting the state of the repository just after the application of it. We use the **git checkout** command for that.

Visiting a commit

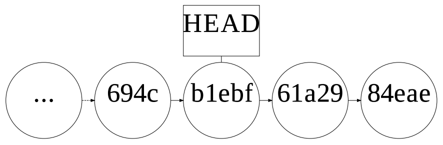
The **`git checkout`** command can be used to take the repo to the state it was just after the application of a certain commit. Get into the UAMMD repo you cloned before and use the log command to choose a particular commit, for instance 61a299, then run:

```
$ git checkout 61a299
```

You will probably see git warning you about being in a "detached HEAD state". This is related to branches, a concept we have not discussed yet. Ignore it for now. See where HEAD is pointing now by using git log. Try to go to the previous commit using

```
$ git checkout HEAD~1
```

Look for HEAD in git log again. You will end up with something like:



You will notice that once you checkout a commit git will always complain about being in a "detached HEAD", even if you go back to the original commit. Detached here refers to a branch, in other words your repo is now "detached from any branch". To understand what this means we need to talk about branches.

But before we go to branches, we are still lacking the power to modify the repository by adding a commit ourselves. Lets talk about that.

2.3 Creating commits

We are going to introduce some change in the repository we created before and append a new commit to it. Go back to the myRepo directory, where we created a file called README.md. Creating a commit has two steps:

1. Make git take the changes into account for the next commit (creating or deleting a file is a change). We use the add command for this.

2. Pack the changes into a commit. We use the commit command for this.

When we want to upload some new commits to a remote more steps are required, but for the moment our new repo has no remotes.
First, lets ask git about the current status of the repo.

Querying the current status of the repo

The status command will prompt useful information about the current state of the repo, such as the current commit, the modified files or the current branch. It will also tell us if there is currently some kind of inconsistency in the repository

```
$ git status
```

There are a lot of ways to customize how status feeds you its information, try the help command!
Pay close attention to the status output, git is very informative and its advice on how to proceed is usually really good.

Try to run git status in the newly created repo. It will tell you that:

- You are in the branch called "master".
- There are no commits in this repo.
- There are Untracked files.

2.3.1 Types of file in a repository

Inside the folder structure of a repository a given file can be either tracked (meaning that git is aware of its existence) or untracked (a file that is not part of the repository, not handled by git).
When a tracked file is modified git will recognize it, opening a new distinction. Changes to a tracked file can be either staged or unstaged. In other words, whether git acknowledges the changes or not. An untracked file is made tracked in the same way that an unstaged change is made staged, by using the **git add** command.

Staging changes

Making git aware of a new file (untracked file) or of changes to an existing file (staging) is done with the add command:

```
$ git add [files]
```

Sometimes you may want to stage only a portion of a modified file, you can pass the -p option to add, which will prompt you with the modified parts of the file and ask you which of them you want to stage.

There are files that you typically do not want to include in repos, but tend to pollute the folder. For instance, temporal UNIX files that end in ~ or latex temporal files.
A file called .gitignore in the root of the project will be interpreted by git as a list of rules to ignore. See the one in UAMMD as an example.

Telling git who you are

Git will ask you for your name and email to sign you commits. You can do so with the following commands:

```
$ git config --global
 ↪  user.name "John Doe"
$ git config --global
 ↪  user.email
 ↪  johndoe@example.com
```

2.3.2 Packing changes into a commit

Try to add the README.md file in your new repo and run git status. You will see that the file, which was previously red and marked as untracked is now green under the section "Changes to be committed". So let's create a commit with the changes.

Commiting staged changes

We pack changes into a commit using the commit command:

```
$ git commit
```

If you run it like that you will be prompted with an editor showing you what the changes are and asking for a message describing the changes. Alternatively you can pass the -m flag to include this message automatically.

```
$ git commit -m "Description
↪  of the changes"
```

New commits are placed after HEAD, which is subsequently moved to the new commit.

After doing so you can try to run status and log.

2.4 Branches

The picture of a repository as a concatenated series of commits (that you now know how to play around with) is already quite powerful as a means of bug detecting or tracking history. However, this construct is not that useful when two persons are working asynchronously on the same repo, or when you want to develop some functionality and test it before adding it to the project.
Branches allow the repo's history to be split in two at a commit. It is also possible to merge two branches into one. The principal branch of a repository is typically called "master" or "main". When a remote is present git understands the local and remote versions as two di erent branches (for instance, master and remotes/origin/master).

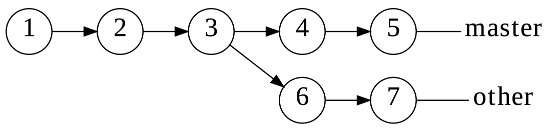


Figure 2: The master branch has been split at the commit 3, starting another branch called "other".

Listing branches in a repo

The git branch command can be used to list branches

```
$ git branch -a
```

Try it in UAMMD, you will see that the current branch is marked in green while all the other ones are called something like "remotes/origin/name" in red. As we discussed, git treats remote branches as different from the local copies, even if they point to the same commit (there is no issue in several branches being on the same commit). When you switch to an existing branch for the first time it will appear in the branch list without the "remotes/origin" part.

Deleting a branch

Confusingly enough, the branch command cannot be used to create a new branch, but it is used to delete one.

```
$ git branch -d name
```

Switch to another branch

To switch to a branch that already exist, simply use checkout.

```
$ git checkout name
```

Note that if you have unstaged changes that would be overwritten by the change of branch git will complain, advising you to either discard them or commit them.

Creating a new branch

To create a new branch called "name" you start by traveling to the commit/branch you want to split from and use the checkout command as:

```
$ git checkout -b [name]
```

This creates a new branch and takes the repo to it (placing HEAD at it), so that any new commits will go into the new branch.

Lets imagine that you have cloned a repo of yours and added some commits to the master branch. In doing so your your master branch has commits that the branch remotes/origin/master does not have, leaving you in a situation similar to figure 3.

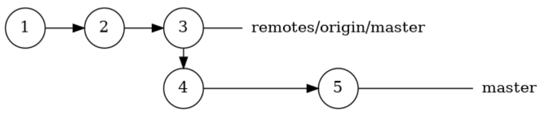


Figure 3: The master branch has evolved past the remotes/origin/master one.

In order to reproduce this situation we need to first create a remote version of our repo and then add it as a remote in our local version. For that, we will have to:

1. Set up an account at GitHub.
2. Create a new repository in GitHub (see instructions here). Choose a public repository and "Initialize from an already existing repository".
3. Add the new remote to your local repo.
4. Push your local branch to the new remote

Adding a remote to a local copy of a repo

We will add a new remote called "origin" to a local copy of a repository. Note that you will typically only need to do this when creating a new repository, as cloning sets up the origin remote automatically. It is useful, though, when you need two remotes for one reason or another.
We will mainly work with GitHub, which o er us a link for our repo (after it has been created) as:

```
$ git remote add origin
↪  https://github.com/USER/R⌋
↪  EPO.git
```

Now you can check the new branch(es) that appeared git the branch command.
The remote command can also be used for many other things, like removing, renaming or changing the url[a] of a remote.

[a]If you have set up TFA in your GitHub account you will not be able to communicate with the remote via the https address, you will need to do so via ssh, which requires to change the remote url to git@github.com:USERNAME/REPOSITORY.git.

Push local commits to a remote branch

Adding new local commits to a remote is called pushing.

```
$ git push [remote] [branch]
```

In newer versions of git, remote defaults to origin and branch to the current one. In general, you might have to write something like:

```
$ git push origin master
```

Pushing will only work if your local version of the branch contains the latest commit of the remote[a]. Otherwise git will have no way of knowing how to reconcile the di erences and you will get an error. In that case you will have to first pull (or fetch + merge) to synchronize your local branch with the remote (ensuring that the latest commit in the remote is also in your local branch) and then push.

[a]There is the special situation in which there are no commits in the remote, in which case pushing will just populate the branch with the new commits.

Update the remote branches in your local copy

We use the fetch command to make a local copy aware of changes in remote branches. Note that fetch will simply advance the remotes/origin/ branches (see git branch -a), it will leave the local ones unmodified.

```
$ git fetch
```

Showing the di erence between two commits

Sometimes it is useful to list the di erences between two commits instead of git showing a single one. We can use the di command for that

```
$ git diff [commit] [commit]
```

The di command is quite powerful and has a lot of forms, for instance

```
$ git diff [commit]...[commit]
```

Will show the the di erences between one commit and every other, including up to, the second one.

As long as you are the only contributor to the repo and you are contributing only using the same local copy every time then this workflow will be enough. However, when you are working with a remote Git repository, it is possible that other collaborators will push new commits to the repository while you are working on your local copy (see 4). If this happens, your local repository will no longer be up to date with the remote repository, and you will need to incorporate the new changes from the remote repository into your local copy before you can push your own changes.

If you were to run the git push command in this situation you would be prompted with an error similar to the following:

```
$ git push origin master
  To https://github.com/user/repo.git
  ! [rejected]        master -> master
  ↪  (non-fast-forward)
```

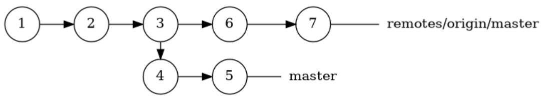


Figure 4: The local master branch its remote counterpart (remotes/origin/master) have diverged. An equivalent picture could be produced if we create two local branches, one could be the master branch and another a feature branch, in which we are testing some new functionality.

```
error: failed to push some refs to '
↪  https://github.com/user/repo.git'
hint: Updates were rejected because
↪  the tip of your current branch
↪  is behind
hint: its remote counterpart.
↪  Integrate the remote changes
```

Git cannot push your commits because there are already some other commits after the point where your local ones should go. In other words, the remote and the local copies of the master branch have diverged. We have to somehow merge both branches. In this situation we would have to first make our local copy aware of the new commits in the remote by using the git fetch command and then use the git merge command to unify both.

Merging changes from one branch into another

When working with Git, it is often necessary to merge changes from one branch into another. This can be useful when you want to combine the work you have done in one branch with the work of other collaborators in another branch, or when you want to incorporate changes from a remote branch into your local branch.

The git merge command allows you to do this. To use the command, you specify the branch you want to merge and the branch you want to merge into, like this:

```
$ git merge [branch-to-merge]
↪  [branch-to-merge-into]
```

For example, if you have made some changes on your local feature branch and want to merge them into the main branch, you can use the following command:

```
$ git merge feature master
```

This will combine the changes from the feature branch into the master branch. If there are no conflicts between the changes in the two branches, git merge will automatically merge the changes and create a new commit to record the merge. However, if there are conflicts, git merge will stop and allow you to resolve the conflicts manually before continuing.

Most of the time git will make a good job merging branches but there are situations in which conflicts will appear between two branches that require human interaction. For instance, when two collaborators have modified the same line of the same file, or when files have been moved around. In these cases git merge will give you a list of files that require manual handling. To resolve the conflicts, you will need to edit the a ected files and remove the conflict markers that Git has inserted. Once you have resolved all the conflicts, you can use the git add command to stage the changes and then use the git commit command to commit the changes.

Updating your local repository with changes from a remote repository

The git pull command is equivalent to running git fetch followed by git merge. To use the git pull command, you simply specify the remote repository and the branch you want to update, like this:

```
$ git pull [remote] [branch]
```

For example, if you want to update your local master branch with the latest changes from the origin remote, you can use the following command:

```
$ git pull origin master
```

This will fetch the latest changes from the origin/master branch and merge them into your local master branch.

3 Welcome to (git) hell

Git hell is a situation that can arise when working with Git, where you are unable to merge or push changes because of conflicts or other issues. This can happen when multiple collaborators are working on the same codebase and making conflicting changes, or when you have made changes that are not compatible with the latest version of the code in the remote repository.

Here is an example of a scenario that could lead to Git hell:

1. You are working on a feature branch on your local repository and make several commits to your branch.

2. Meanwhile, your collaborator makes changes to the master branch and pushes them to the remote repository.

3. When you try to push your changes to the remote repository, you are unable to do so because the master branch has changed since you last pulled from the remote repository.

4. You try to use git pull to update your local repository with the latest changes from the remote repository, but there are conflicts between your changes and the changes your collaborator has made.

5. You are unable to resolve the conflicts automatically using git pull, so you have to manually resolve the conflicts.

6. After resolving the conflicts, you try to push your changes again, but the remote repository has changed again since you last pulled, so there are more conflicts.

In this scenario, you are unable to push your changes to the remote repository because of conflicts between your local changes and the changes made by your collaborator. This can be frustrating and time-consuming, and can lead to a situation where you are unable to make progress on your work. This is an example of Git hell. Nonetheless, if you have followed the lessons and the cards up until now, you have all the necessary tools to gat out of git hell.

4 Advanced functionality

4.1 Cherry-picking

Cherry-picking specific commits from one branch and applying them to another

The git cherry-pick command allows you to select specific commits from one branch and apply them to another branch, without merging the entire branch [a].

To use the git cherry-pick command,

you specify the commit hash of the commit you want to cherry-pick, like this:

```
$ git cherry-pick [commit]
```

[a]You could mimic this behavior with a combination of checkout, branch and merge, tools you already know

This will apply the changes from the specified commit to your current branch. If there are conflicts between the changes in the commit and the changes in your current branch, git will warn you and allow you to resolve the conflicts manually.

4.2 Stashing

Stashing your changes to switch to a di erent branch

The git stash command allows you to save your changes temporarily and switch to a di erent branch. This can be useful when you are working on a branch and need to switch to a di erent branch quickly, but you don't want to commit your changes or discard them.

To use the git stash command, you simply run the command without any arguments, like this:

```
$ git stash
```

This will save your changes to a temporary stash and reset your working tree to the latest commit on your current branch. You can then switch to a di erent branch and work on that branch, and your changes will be saved in the stash.

When you are ready to restore your changes from the stash, you can use the git stash pop command, which will apply the changes from the stash and remove the stash from the list of stashes. For example:

```
$ git stash pop
```

This will apply the changes from the stash and remove the stash from the list of stashes. You can also use the git stash list command to view a list of stashes and the git stash apply command to apply the changes from a stash without removing it from the list.

4.3 Rebasing

Imagine the following situation:

1. You are working on a feature branch and you have made several commits to the branch.
2. Meanwhile, other collaborators have made changes to the master branch and pushed them to the remote repository.
3. You want to incorporate the latest changes from the master branch into your feature branch to reduce the number of conflicts you will have to resolve when you try to merge your branch into master.

To do this, you can run the git rebase command. This will move or combine the commits on the feature branch so that they are based on the latest commits in the master branch. If there are conflicts between the two branches, git will warn you and allow you to resolve the conflicts manually. After running the git rebase command, your feature branch will be more up-to-date and will have fewer conflicts with the master branch, making it easier to merge your branch into master when you are ready.

Rebase: Modifying the history of a branch

The git rebase command allows you to modify the history of a branch by moving or combining commits. This can be useful for cleaning up your branch history and making it easier to read, but it can also be dangerous because it can cause conflicts and can cause you to lose work if not used carefully.

To use the git rebase command, you specify the branch you want to rebase and the branch you want to rebase onto, like this:

```
$ git rebase [branch]
↪   [onto-branch]
```

For example, if you want to rebase your feature branch onto the master branch, you can use the following command:

```
$ git rebase feature master
```

This will move or combine the commits on the feature branch so that they are based on the latest commits in the master branch. If there are conflicts between the two branches, git will warn you and allow you to resolve the conflicts manually. Note that this means that the commit hashes in the feature branch are invalidated and e ectively created again. Thus, you are changing history. You should never rebase commits that have left your local copy at some point.

The git rebase command is a powerful tool that allows you to modify the history of a branch. Use it carefully and only when you are familiar with the implications of rebasing, as it can cause conflicts and can cause you to lose work if not used correctly.