UniTs - University of Trieste

Faculty of Data Science and Artificial Intelligence/Scientific Computing

Department of mathematics informatics and geosciences

Generative AI and NLP

Author:
Christian Faccio

March 9, 2025

github.com/christianfaccio christianfaccio@outlook.it

Contents

1 Git

GitHub Cheatsheet

1.1 About Git

1.1.1 About version control and Git

Definition: *Version control*

A version control system, or VCS, tracks the history of changes as people and teams collaborate on projects together. As developers make changes to the project, any earlier version of the project can be recovered at any time.

Developers can revew project history to find out:

- Which changes were made?
- Who made the changes?
- When were the changes made?
- Why were changes needed?

In a **distributed version control system (DVCSs)**, every developer has a full copy of the prokect and project history. They don't need a constant connection to a central repository. `Git` is the most popular distributed version control syste. It is commonly used for both open source and commercial software development, with significant benefits for individuals, teams and businesses.

Tip:

- **Git lens** is used to see the entire timeline of the project, including decisions, changes and progressions. From the moment they access the history of a project, the developer has all the context they need to understand it and start contributing.
- Developers work in every time zone. With a DVCS like Git, collaboration can happen any time while maintaining source code integrity. Using **branches**, developers can safely propose changes to production code.

1.1.2 Repositories

A repository, or Git project, encompasses the entire collection of files and folders associated with a project, along with each file's revision history. The file history appears as **snapshots** in time called commits. The commits can be organized into multiple lines of development called branches. Because Git is a DVCS, repositories are self-contained units and anyone who has a copy of the repository can access the entire codebase and its history. Using the command line or other ease-of-use interfaces, a Git repository also allows for: interaction with the history, cloning the repository, creating branches, committing, merging, comparing changes across versions of code, and more.
A repository, or Git project, encompasses the entire collection of files and folders associated with a project, along with each file's revision history. The file history appears as snapshots in time

called commits. The commits can be organized into multiple lines of development called branches. Because Git is a DVCS, repositories are self-contained units and anyone who has a copy of the repository can access the entire codebase and its history. Using the command line or other ease-of-use interfaces, a Git repository also allows for: interaction with the history, cloning the repository, creating branches, committing, merging, comparing changes across versions of code, and more.

1.1.3 GitHub

GitHub hosts Git repositories and provides developers with tools to ship better code through command line features, issues (threaded discussions), pull requests, code review, or the use of a collection of free and for-purchase apps in the GitHub Marketplace. With collaboration layers like the GitHub flow, a community of 100 million developers, and an ecosystem with hundreds of integrations, GitHub changes the way software is built.

GitHub builds collaboration directly into the development process. Work is organized into repositories where developers can outline requirements or direction and set expectations for team members. Then, using the GitHub flow, developers simply create a branch to work on updates, commit changes to save them, open a pull request to propose and discuss changes, and merge pull requests once everyone is on the same page.

1.1.4 Command Line Interface

To use Git, developers use specific commands to copy, create, change, and combine code. These commands can be executed directly from the command line or by using an application like GitHub Desktop. Here are some common commands for using Git:

- `git init`: Initializes a new Git repository. Until you run this command inside a repository or directory, it's just a regular folder. Only after you input this does it accept further Git commands.
- `git config`: Short for "configure," this is most useful when you're setting up Git for the first time.
- `git help`: Forgot a command? Type this into the command line to bring up the 21 most common git commands.
- `git status`: We all get a little nervous the first time we commit our changes. This command shows you the status of changes as untracked, modified, or staged.
- `git add`: This does not add new files to your repository. Instead, it brings new files to Git's attention. After you add files, they're included in Git's "snapshots" of the repository.
- `git commit`: Git's most important command. After you make any sort of change, you input this in order to take a "snapshot" of the repository. Usually it goes `git commit -m "Message here"`.
- `git branch`: Working on multiple features at once? Use this command to list, create, or delete branches.
- `git checkout`: Literally allows you to "check out" a repository that you are not currently inside. This is a navigational command that lets you move to the repository you want to check.
- `git merge`: When you're done working on a branch, you can merge your changes back to the master branch, which is visible to all collaborators.
- `git push`: If you're working on your local computer, and want your commits to be visible online on GitHub as well, you "push" the changes up to GitHub with this command.
- `git pull`: If you're working on your local computer and want your commits to be visible online on GitHub as well, you "push" the changes up to GitHub with this command.

- `git clone`: If you're starting fresh and want to clone an existing repository from GitHub to your local computer, you can use this command to copy the repository to your computer.
- `git remote`: When you clone a repository from GitHub, it automatically creates a connection or "remote" that you can also push changes to.

Example: Contribute to an existing repository

```
# download a repository on GitHub to our machine
# Replace `owner/repo` with the owner and name of the repository to
    clone
git clone https://github.com/owner/repo.git

# change into the `repo` directory
cd repo

# create a new branch to store any new changes
git branch my-branch

# switch to that branch (line of development)
git checkout my-branch

# make changes, for example, edit `file1.md` and `file2.md` using the
    text editor

# stage the changed files
git add file1.md file2.md

# take a snapshot of the staging area (anything that's been added)
git commit -m "my snapshot"

# push changes to github
git push --set-upstream origin my-branch
```

Example: Start a new repository adn publish on GitHub

```
# create a new directory, and initialize it with git-specific functions
git init my-repo

# change into the `my-repo` directory
cd my-repo

# create the first file in the project
touch README.md

# git isn't aware of the file, stage it
git add README.md

# take a snapshot of the staging area
git commit -m "add README to initial commit"

# provide the path for the repository you created on github
git remote add origin https://github.com/YOUR-USERNAME/YOUR-REPOSITORY-
    NAME.git

```

```
# push changes to github
git push --set-upstream origin main
```

Example: Contribute to an existing branch on GitHub

```
# change into the `repo` directory
cd repo

# update all remote tracking branches, and the currently checked out
    branch
git pull

# change into the existing branch called `feature-a`
git checkout feature-a

# make changes, for example, edit `file1.md` using the text editor

# stage the changed file
git add file1.md

# take a snapshot of the staging area
git commit -m "edit file1"

# push changes to github
git push
```

1.1.5 Models for collaborative development

THere are two primary ways people collaborate on GitHub:

- **Fork and pull model**: A project owner creates a master repository, and contributors fork that repository to their accounts. They clone the repository to their local machine, make changes, commit them
- **Shared repository model**: In this model, all collaborators have push access to the same repository. This is more common for small teams and organizations.

With a shared repository, individuals and teams are explicitly designated as contributors with read, write, or administrator access. This simple permission structure, combined with features like protected branches, helps teams progress quickly when they adopt GitHub.

For an open source project, or for projects to which anyone can contribute, managing individual permissions can be challenging, but a fork and pull model allows anyone who can view the project to contribute. A fork is a copy of a project under a developer's personal account. Every developer has full control of their fork and is free to implement a fix or a new feature. Work completed in forks is either kept separate, or is surfaced back to the original project via a pull request. There, maintainers can review the suggested changes before they're merged.

1.2 Pushing commits to a remote repository

1.2.1 About `git push`

The `git push` command takes two arguments.

- A remote name, for example, `origin`
- A branch name, for example, `main`

For example, the command `git push origin main` pushes the commits in the local `main` branch to the remote repository named `origin`.

Example: *Pushing changes to a remote repository*

```
# push changes to the remote repository
git push origin main
```

1.2.2 Renaming branches

To rename a branch, you'd use the same `git push` command, but you would add one more argument: the name of the new branch. For example:

Example: *Renaming a branch*

```
# rename the local branch to new-name
git push origin main:new-name
```

This pushes the `main` branch to the remote repository, but the branch is renamed to `new-name`.

Tip:

- If you want to delete a branch, you can use the `--delete` flag with the `git push` command. For example, `git push origin --delete new-name` will delete the `new-name` branch from the remote repository.
- If you want to rename the branch you're currently on, you can use the `-u` flag to set the upstream branch. For example, `git push -u origin main:new-name` will push the `main` branch to the remote repository, renaming it to `new-name`, and set the upstream branch to `new-name`.

1.2.3 Dealing with ”non-fast-forward” errors

If your local copy of a repository is out of sync with, or ”behind”, the upstream repository you’re pushing to, you’ll get a message saying `non-fast-forware updates were rejected`. This means that you must retrieve, or ”fetch,” the upstream changes, before you are able to push your local changes.

> **Tip**: ***Fetching***
>
> Fetching means retrieving recent commits from a remote repository without merging them into your local branch. This lets you view and compare new changes before deciding how to incorporate them into your work.

To fetch the changes from the remote repository, you can use the `git fetch` command. This command retrieves the changes from the remote repository, but it doesn’t merge them into your local branch. After fetching the changes, you can merge them into your local branch using the `git merge` command.

1.2.4 Pushing tags

By default, and without additional parameters, `git push` sends all matching branches that have the same names as remote branches.
To push a single tag, you cna issue the same command as pushing a branch:

```
1 git push origin tag-name
```

To push all your tags, you can type the command:

```
1 git push origin --tags
```

1.2.5 Deleting a remote branch or tag

The syntax to delete a branch is a bit more arcane at first glance:

```
1 git push origin --delete branch-name
```

Note that there is a space before the colon. The command resembles the same steps you’d take to rename a branch. However, here, you’re telling Git to push **nothing** into `branch-name`, effectively deleting it. Because of this, `git push` deletes the branch on the remote repository.

1.2.6 Remotes and forks

You can **fork a repository** on GitHub.
When you clone a repository you own, you provide it with a remote URL that tells Git where to fetch and push updates. If you want to collaborate with the original repository, you’d add a new remote URL, typically called `upstream`, to your local Git clone:

```
1 git remote add upstream REMOTE_URL
```

Now you can fetch updates and branches from their fork:

```
git fetch upstream
# Grab the upstream remote's branches
> remote: Counting objects: 75, done.
> remote: Compressing objects: 100% (53/53), done.
> remote: Total 62 (delta 27), reused 44 (delta 9)
> Unpacking objects: 100% (62/62), done.
> From https://github.com/OCTOCAT/REPO
>  * [new branch]      main     -> upstream/main
```

When you're done making local changes, you can push your local branch to GitHub and initiate a pull request.

1.3 Getting changes from a remote repository

1.3.1 Options for getting changes

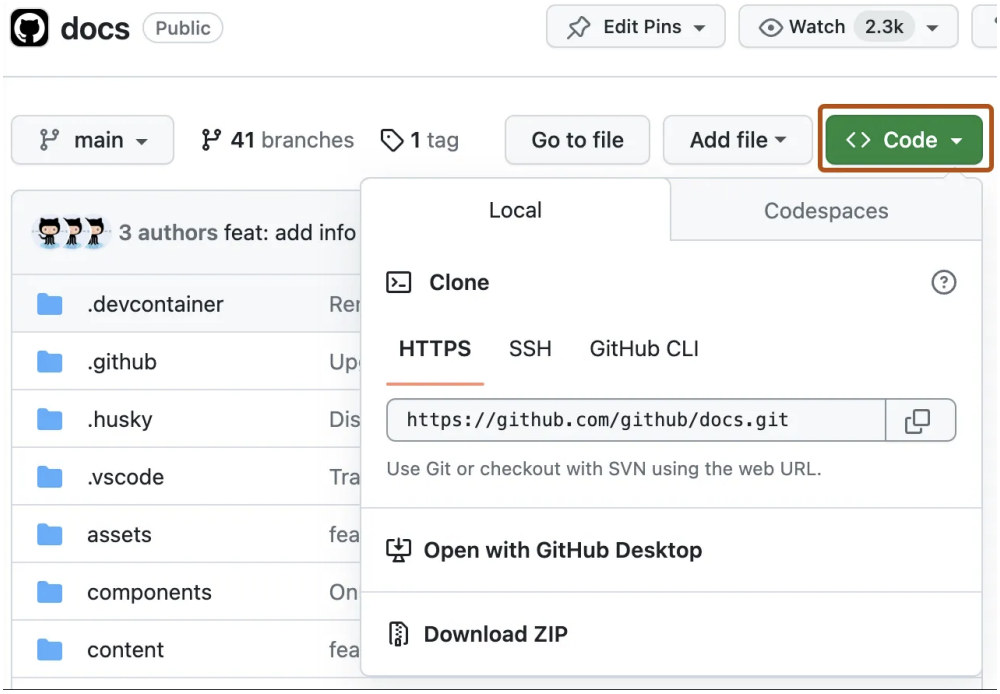
These commands are very useful when interacting with a remote repository. `clone` and `fetch` download remote code from a repository's remote URL to your local computer, `merge` is used to merge different people's work together with yours, and `pull` is a combination of `fetch` and `merge`.

1.3.2 Cloning a repository

To grab a complete copy of another user's repository, use `git clone` like this:

```
git clone https://github.com/USERNAME/REPOSITORY.git
# Clones a repository to your computer
```

You can choose from different URLs when cloning a repository. While logged in to GitHub, there URLs are available on the main page of the repository when you click `<> Code`.



When you run `git clone`, the following actions occur:

- A new folder called `repo` is made
- It is initialized as a Git repository
- A remote named `origin` is created, pointing to the URL you cloned from
- All of the repository's files and commits are downloaded there
- The default branch is checked out

For every branch `foo`, a corresponding remote-tracking branch `refs/remotes/origin/foo` is created in your local repository. You can usually abbreviate such remote-tracking branch names to `origin/foo`.

1.3.3 Fetching changes from a remote repository

Use `fit fetch` to retrieve new work done by other people. Fetching from a repository grabs all the new remote-tracking branches and tags without merging those changes into your own branches. If you already have a local repository with a remote URL set up for the desired project, you can grab all the new information by using `fit fetch *remotename*` in the terminal.

```
1 git fetch REMOTE-NAME
2 # Fetches updates made to a remote repository
```

1.3.4 Merging changes into your local branch

Merging combines your local changes with changes made by others.
Typically, you'd merge a remote-tracking branch (i.e., a branch fetched from a remote repository) with your local branch:

```
1 git merge REMOTE-NAME/BRANCH-NAME
2 # Merges updates made online with your local work
```

1.3.5 Pulling changes from a remote repository

`git pull` is a convenient shortcut for completing both `git fetch` and `git merge` in the same command:

```
1 git pull REMOTE-NAME BRANCH-NAME
2 # Grabs online updates and merges them with your local work
```

Because `pull` performs a merge on the retrieved changes, you should ensure that your local work is committed before running the `pull` command. If you run into a **merge conflict** you cannot resolve, or if you decide to quit the merge, you can use `git merge --abort` to take the branch back to where it was in before you pulled.

1.4 Dealing with non-fast-forward errors

Sometimes, Git can't make your change to a remote repository without losing commits. When this happens, your push is refused.
If another person has pushed to the same branch as you, Git won't be able to push your changes:

Example:

```
git push origin main
> To https://github.com/USERNAME/REPOSITORY.git
>  ! [rejected]        main -> main (non-fast-forward)
> error: failed to push some refs to 'https://github.com/USERNAME/
    REPOSITORY.git'
> To prevent you from losing history, non-fast-forward updates were
     rejected
> Merge the remote changes (e.g. 'git pull') before pushing again.
    See the
> 'Note about fast-forwards' section of 'git push --help' for
    details.
```

You can fix this by fetching and merging the changes made on the remote branch with the changes that you have made locally:

```
git fetch origin
# Fetches updates made to an online repository

git merge origin YOUR_BRANCH_NAME
# Merges updates made online with your local work
```

Or simply use `git pull` to perform both commands at once:

```
git pull origin YOUR_BRANCH_NAME
# Grabs online updates and merges them with your local work
```

1.5 Splitting a subfolder out into a new repository

You can turn a folder within a Git repository into a brand new repository.

Warning:

You need Git version 2.22.0 or later to follow these instructions, otherwise `git filter-repo` will not work.

If you create a new clone of the repository, you won't lose any of your Git history or changes when you split a folder into a separate repository. However, note that the new repository won't have the branches and tags of the original repository.
Steps:

1. Open Terminal.
2. Change the current working directory to the location where you want to create your new repository.
3. Clone the repository that contains the subfolder.

```
git clone https://github.com/USERNAME/REPOSITORY-NAME
```

4. Change the current working directory to your cloned repository.

```
cd REPOSITORY-NAME
```

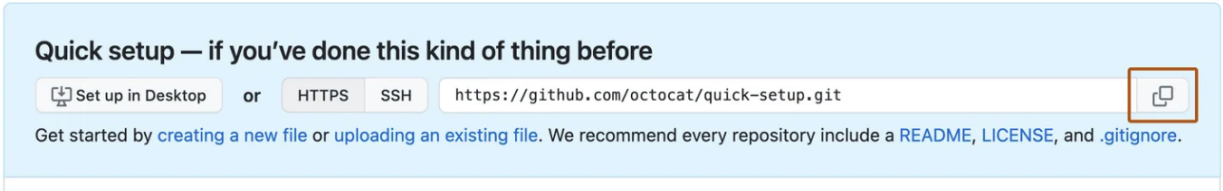
5. To filter out the subfolder from the rest of the files in the repository, install `git-filter-repo`, then run `git filter-repo` with the following arguments.
   - `FOLDER-NAME`: The folder within your project where you'd like to create a separate repository.

```
git filter-repo --path FOLDER-NAME/
# Filter the specified branch in your directory and remove empty
    commits
```

   The repository should now only contain the files that were in your subfolder(s).
   If you want one specific subfolder to be the new root folder of the new repository, you can use the following command:

```
git filter-repo --subdirectory-filter FOLDER-NAME/
# Filter the specified branch in your directory and remove empty
    commits
```

6. Create a repository on GitHub.
7. At the top of your new repository on GitHub's Quick Setup page, click ☐☐ to copy the remote repository URL.



Tip:

For information on the difference between HTTPS and SSH URLs, see About remote repositories.

8. Add a new remote name with the URL you copied for your repository. For example, `origin` or `upstream` are two common choices.

```
git remote add NEW-REMOTE-NAME URL
```

9. Verify that the remote URL was added with your new repository name.

```
git remote -v
# Verify new remote URL
> origin  https://github.com/USERNAME/NEW-REPOSITORY-NAME.git (
    fetch)
```

```
4 > origin  https://github.com/USERNAME/NEW-REPOSITORY-NAME.git (push
      )
```

10. Push your changes to the new repository on GitHub.

```
1 git push -u origin BRANCH-NAME
```

1.6 About Git subtree merges

Typically, a subtree merge is used to contain a repository within a repository. The ”subrepository” is stored in a folder of the main repository.
The best way to explain subtree merges is to show by example. We will:

- Make an empty repository called `test` that represents our project.
- Merge another repository into it as a subtree colled `Spoon-Knife`.
- The `test` project will use that subproject as if it were part of the same repository.
- Fetch updates from `Spoon-Knife` into our `test` project.

1.6.1 Setting up the empty repository for a subtree merge

1. Open Terminal.
2. Create a new directory and navigate to it.

```
1 mkdir test
2 cd test
```

3. Initialize the directory as a Git repository.

```
1 git init
2 > Initialized empty Git repository in /Users/octocat/tmp/test/.git/
```

4. Create and commit a new file.

```
1 touch .gitignore
2 git add .gitignore
3 git commit -m "initial commit"
4 > [main (root-commit) 3146c2a] initial commit
5 >  0 files changed, 0 insertions(+), 0 deletions(-)
6 >  create mode 100644 .gitignore
```

1.6.2 Adding a new repository as a subtree

1. Add a new remote URL pointing to the separate project that we’re interested in.

```
1 git remote add -f spoon-knife https://github.com/octocat/Spoon-
     Knife.git
2 > Updating spoon-knife
3 > warning: no common commits
4 > remote: Counting objects: 1732, done.
5 > remote: Compressing objects: 100% (750/750), done.
6 > remote: Total 1732 (delta 1086), reused 1558 (delta 967)
```

```
> Receiving objects: 100% (1732/1732), 528.19 KiB | 621 KiB/s, done
    .
> Resolving deltas: 100% (1086/1086), done.
> From https://github.com/octocat/Spoon-Knife
>  * [new branch]      main     -> Spoon-Knife/main
```

2. Merge the `Spoon-Knife` project into the local Git project. This does not change any of your files locally, but it does prepare Git for the next step.
   If you're using Git 2.9 or above:

```
git merge -s ours --no-commit --allow-unrelated-histories spoon-
    knife/main
> Automatic merge went well; stopped before committing as requested
```

   If you're using Git 2.8 or below:

```
git merge -s ours --no-commit spoon-knife/main
> Automatic merge went well; stopped before committing as requested
```

3. Create a new directory called **spoon-knife**, and copy the Git history of the `Spoon-Knife` project into it.

```
git read-tree --prefix=spoon-knife/ -u spoon-knife/main
> fatal: refusing to merge unrelated histories
```

4. Commit the changes to keep them safe.

```
git commit -m "Subtree merged in spoon-knife"
> [main fe0ca25] Subtree merged in spoon-knife
```

Although we've only added one subproject, any number of subprojects can be incorporated into a Git repository.

Tip:

If you create a fresh clone of the repository in the future, the remotes you've added will not be created for you. You will have to add them again using the `git remote add` command.

1.6.3 Synchronizing with updates and changes

When a subproject is added, it is not automatically kept in sync with the upstream changes. You will need to update the subproject with the following command:

```
git pull -s subtree REMOTE-NAME BRANCH-NAME
```

For the example above, this would be:

```
git pull -s subtree spoon-knife main
```

1.7 About Git rebase

Typically, you would use `git rebase` to:

- Edit previous commit messages
- Combine multiple commits into one
- Delete or revert commits that are no longer necessary

Warning:

Because changing your commit history can make things difficult for everyone else using the repository, it's considered bad practice to rebase commits when you've already pushed to a repository. To learn how to safely rebase, see About pull request merges.

1.7.1 Rebasing commits against a branch

To rebase all the commits between another branch and the current branch state, you can enter the following command in your shell (either the command prompt for Windows, or the terminal for Mac and Linux):

```
1 git rebase --interactive OTHER-BRANCH-NAME
```

1.7.2 Rebasing commits against a point in time

To rebase the last few commits in your current branch, you can enter the following command in your shell:

```
1 git rebase --interactive HEAD~7
```

1.7.3 Commands available while rebasing

There are six commands available while rebasing:

- `pick`
  simply means that the commits is included. Rearranging the order of the `pick` commands changes the order of the commits when the rebase is underway. If you choose not to include a comit, you should delete the entire line.
- `reword`
  it is similar to `pick`, but after you use it, the rebase process will pause and give you a chance to alter the commit message. Any changes made by the commit are not affected.
- `edit`
  if you choose to `edit` a commit, you'll be given the chance to amend the commit, meaning that you can add or change the commit entirely. you can also make more commits before you continue the rebase. This allows you to split a large commit into smaller ones, or, remove erroneus changes made in a commit.
- `squash`
  it lets you combine two or more commits into a single one. A commit is squashed into the commit above it. Git gives you the chance to write a new commit message describing both changes.
- `fixup`
  similar to `squash`, but the commit to be merged has its message discarded. The commit is

simply merged into the commit above it, and the earlier commit's message is used to describe both changes.

- `exec`
  This lets you run arbitrary shell commands against a commit.

1.7.4 An example of using `git rebase`

No matter which command you use, Git will launch your default editor and open a file that details the commits in the range you've chosen. That file looks something like this:

Example:

```
pick 1fc6c95 Patch A
pick 6b2481b Patch B
pick dd1475d something I want to split
pick c619268 A fix for Patch B
pick fa39187 something to add to patch A
pick 4ca2acc i cant' typ goods
pick 7b36971 something to move before patch B

# Rebase 41a72e6..7b36971 onto 41a72e6
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

Breaking this information from top to bottom, we see that:

- Seven commits are listed, which indicates that there were seven changes between out starting point and our current branch state.
- The commits you chose to rebase are sorted in the order of the oldest changes (at the top) to te newest changes (at the botom).
- Each line lists a command (by default, `pick`), the commit SHA, and the commit message. The entire `git rebase` procedure centers around your manipulation of these three columns. The changes you make are rebased onto your repository.
- After the commits, Git tells you the range of commits we're working with (`41a72e6..7b36971`).
- Finally, Git gives some help by telling you the commands that are available to you when rebasing commits.

1.8 Using Git rebase on the command line

Here, all of the `git rebase` commands available, exept for `exec`, are covered.

We'll start our rebase by entering `git rebase --interactive HEAD 7` on the terminal. Our favourite text editor will display the following lines:

```
pick 1fc6c95 Patch A
pick 6b2481b Patch B
pick dd1475d something I want to split
pick c619268 A fix for Patch B
pick fa39187 something to add to patch A
pick 4ca2acc i cant' typ goods
pick 7b36971 something to move before patch B
```

In this example, we're going to:

- Squash the fifth commit (`fa39187`) into the `"Patch A"` commit (`1fc6c95`), using `squash`.
- Move the last commit (`7b36971`) up before the `"Patch B"` commit (`6b2481b`), and keep it as `pick`.
- Merge the `"A fix for Patch B"` commit (`c619268`) into the `"Patch B"` commit (`6b2481b`), and disregard the commit message using `fixup`.
- Split the third commit (`dd1475d`) into two smaller commits, using `edit`.
- Fix the commit message of the misspelled commit (`4ca2acc`), using `reword`.

This sounds like a lot of work, but by taking it one step at a time, we can easily make those changes. To start, we'll need to modify the commands in the file to look like this:

```
pick 1fc6c95 Patch A
squash fa39187 something to add to patch A
pick 7b36971 something to move before patch B
pick 6b2481b Patch B
fixup c619268 A fix for Patch B
edit dd1475d something I want to split
reword 4ca2acc i cant' typ goods
```

We've changed each line's command from `pick` to the command we're interested in.

Now, save and close the editor; this will start the interactive rebase.

Git skips the first rebase command, `pick 1fc6c95`, since it does not need to do anything. It goes to the next command, `squash fa39187`. Since this operation requires your input, Git opens your text editor once again. The file it opens up looks something like this:

```
# This is a combination of two commits.
# The first commit's message is:

Patch A

# This is the 2nd commit message:

something to add to patch A

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

```
# Not currently on any branch.
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   a
#
```

This file is Git's way of saying, "Hey, here's what I'm about to do with this `squash`."It lists the first commit's message (`"Patch A"`), and the second commit's message (`"something to add to patch A"`). If you're happy with these commit messages, you can save the file, and close the editor. Otherwise, you have the option of changing the commit message by simply changing the text.
When the editor is closed, the rebase continues:

```
ù
pick 1fc6c95 Patch A
squash fa39187 something to add to patch A
pick 7b36971 something to move before patch B
pick 6b2481b Patch B
fixup c619268 A fix for Patch B
edit dd1475d something I want to split
reword 4ca2acc i cant' typ goods
```

Git processes the two `pick` commands (for `pick 7b36971` and `pick 6b2481b`). It also processes the `fixup` command (`fixup c619268`), since it does not require any interaction. `fixup` merges the changes from `c619268` into the commit before it, `6b2481b`. Both changes will have the same commit message: `"Patch B"`.
Git gets to the `edit dd1475d` operation, stps, and prints the following message to the terminal:

```
You can amend the commit now, with

    git commit --amend

Once you are satisfied with your changes, run

    git rebase --continue
```

At this pount, you can edit eny of the files in your project to make any additional changes. For each change you make, you'll need to perform a new commit, and you can do that by entering the `git commit --amend` command. When you're finished making all your changes, you can run `git rebase --continue`.
Git then gets to the `reword 4ca2acc` command. It opens up your text editor one more time, and presents, the following information:

```
i cant' typ goods

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# Not currently on any branch.
# Changes to be committed:
```

```
#   (use "git reset HEAD^1 <file>..." to unstage)
#
# modified:   a
#
```

As before, Git is showing the commit message for you to edit. You can change the text (`"i can't typ goods"`), save the file, and close the editor. Git will finish the rebase and return you to the terminal.

1.8.1 Pushing rebased code to GitHub

Since you've altered Git history, the usual `git push origin` will not work. You'll need to modify the command by "force-pushing" your latest changes:

```
# Don't override changes
git push origin main --force-with-lease

# Override changes
git push origin main --force
```

Warning:

Force pushing has serious implications because it changes the historical sequence of commits for the branch. Use it with caution, especially if your repository is being accessed by multiple people.

1.9 Resolving merge conflicts after a Git rebase

When you perform a `git rebase` operation, you're typically moving commits around. Because of this, you might get into a situation where a merge conflict is introduced. That means that two of your commits modified the same line in the same file, and Git does not know which change to apply.
After you reorder and manipulate commits using `git rebase`, should a merge conflict occur, Git will tell you so whith the following message in the terminal:

```
error: could not apply fa39187... something to add to patch A

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --
    abort".
Could not apply fa39187f3c3dfd2ab5faa38ac01cf3de7ce2e841... Change fake
    file
```

Here, Git is telling you which commit is causing the conflict (`fa39187`). You're given three choices:

- You can run `git rebase --abort` to completely undo the rebase. Git will return you to your branch's state as it was before `git rebase` was called.
- You can run `git rebase --skip` to completely skip the commit. That means that none of the changes introduced by the problematic commit will be included. It is very rare that you would

choose this option.

- You can fix the conflict.

To fix the conflict, you can follow the standard procedures for resolving merge conflicts from the command line. When you're finished, you'll need to call `git rebase --continue` in order for Git to continue processing the rest of the rebase.

1.10 Dealing with special characters in branch and tag names

Most repositories use simple branch names, such as `main` or `update-icons`. Tag names also usually follow a basic format, such as a version number like `v1.2.3`. Both branch names and tag names may also use the path separator (`/`) for structure, for example `area/item` or `level-1/level-2/level-3`. Other than some exceptions - such as not starting or ending a name with slash, or having consecutive slashes in the name - Git has very few restrictions on what characters may be used in branch and tag names.

1.10.1 Why you need to escape special characters

When using a CLI, you might have situations where a branch or tag name contains special characters that have a special meaning for your shell environment. To use these characters safely in a Git command, they must be quoted or escaped, otherwise the command may have unintended effects. For example, the `$` character is used by many shells to refer to a variable. Most shells would interpret a valid branch name like `hello-$USER` as equivalent to the word "hello", followed by a hyphen, followed by the current value of the `USER` variable, rather than the literal string `hello-$USER`. If a branch name includes the `$` character, then the shell must be stopped from expanding it as a variable reference. Similarly, if a branch name contains a semi-colon (`;`), most shells interpret it as a command separator, so it needs to be quoted or escaped.

1.10.2 How to escape special characters in branch and tag names

Most branch and tag names with special characters can be handled by including the name in single quotes, for example `'hello-$USER'`.

- In the **Bash** shell, enclosing a string of characters in single quotes preserves the literal value of the characters within the single quotes.
- **Zsh** behaves similar to Bash, however this behavior is configurable using the `RC_QUOTES`
- **PowerShell** also treats characters literally when inside single quotes.

For these shells, the main exception is when the branch or tag name itself contains a single quote. In this case, you should consult the official documentation for your shell.

1.10.3 Naming branches and tags

If possible, create branch and tag names that don't contain special characters, as these would need to be escaped. A safe default set of characters to use for branch names and tag names is:

- The English alphabet(`a` to `z` and `A` to `Z`)
- Numbers (`0` to `9`)
- A limited set of punctuation characters:
  - period (`.`)
  - hyphen (`-`)
  - underscore (`_`)

- forward slash (`/`)

To avoid confusion, you should start branch names with a letter.

1.10.4 Restrictions on names in GitHub

GitHub restricts a small number of branch and tags names from being pushed up. Those restrictions are:

- No names which look like Git object IDs (40 characters containing only 0-9 and A-F), to prevent confusion with actual Git object IDs.
- No names beginning with `refs/`, to prevent confusion with the full name of Git refs.