Git Cheat Sheet

The essential Git commands every developer must know

Table of Content

Creating Snapshots

Initializing a repository

```
git init
```

Staging files

```
git add file1.js             # Stages a single file
git add file1.js file2.js    # Stages multiple files
git add *.js                 # Stages with a pattern
git add .                    # Stages the current directory and all its content
```

Viewing the status

```
git status                   # Full status
git status -s                # Short status
```

Committing the staged files

```
git commit -m “Message”      # Commits with a one-line message
git commit                   # Opens the default editor to type a long message
```

Skipping the staging area

```
git commit -am “Message”
```

Removing files

```
git rm file1.js              # Removes from working directory and staging area
git rm --cached file1.js     # Removes from staging area only
```

Renaming or moving files

```
git mv file1.js file1.txt
```

Viewing the staged/unstaged changes

```
git diff                     # Shows unstaged changes
git diff --staged            # Shows staged changes
git diff --cached            # Same as the above
```

Viewing the history

```
git log                      # Full history
git log --oneline            # Summary
git log --reverse            # Lists the commits from the oldest to the newest
```

Viewing a commit

```
git show 921a2ff             # Shows the given commit
git show HEAD                # Shows the last commit
git show HEAD~2              # Two steps before the last commit
git show HEAD:file.js        # Shows the version of file.js stored in the last commit
```

Unstaging files (undoing git add)

```
git restore --staged file.js # Copies the last version of file.js from repo to index
```

Discarding local changes

```
git restore file.js          # Copies file.js from index to working directory
git restore file1.js file2.js # Restores multiple files in working directory
git restore .                # Discards all local changes (except untracked files)
git clean -fd                # Removes all untracked files
```

Restoring an earlier version of a file

```
git restore --source=HEAD~2 file.js
```

Browsing History

Viewing the history

```
git log --stat              # Shows the list of modified files
git log --patch             # Shows the actual changes (patches)
```

Filtering the history

```
git log -3                  # Shows the last 3 entries
git log --author=“Mosh”
git log --before=“2020-08-17”
git log --after=“one week ago”
git log --grep=“GUI”        # Commits with “GUI” in their message
git log -S“GUI”             # Commits with “GUI” in their patches
git log hash1..hash2        # Range of commits
git log file.txt            # Commits that touched file.txt
```

Formatting the log output

```
git log --pretty=format:”%an committed %H”
```

Creating an alias

```
git config --global alias.lg “log --oneline"
```

Viewing a commit

```
git show HEAD~2
git show HEAD~2:file1.txt   # Shows the version of file stored in this commit
```

Comparing commits

```
git diff HEAD~2 HEAD        # Shows the changes between two commits
git diff HEAD~2 HEAD file.txt # Changes to file.txt only
```

Checking out a commit

```
git checkout dad47ed         # Checks out the given commit
git checkout master          # Checks out the master branch
```

Finding a bad commit

```
git bisect start
git bisect bad               # Marks the current commit as a bad commit
git bisect good ca49180      # Marks the given commit as a good commit
git bisect reset             # Terminates the bisect session
```

Finding contributors

```
git shortlog
```

Viewing the history of a file

```
git log file.txt             # Shows the commits that touched file.txt
git log --stat file.txt      # Shows statistics (the number of changes) for file.txt
git log --patch file.txt     # Shows the patches (changes) applied to file.txt
```

Finding the author of lines

```
git blame file.txt           # Shows the author of each line in file.txt
```

Tagging

```
git tag v1.0                 # Tags the last commit as v1.0
git tag v1.0 5e7a828         # Tags an earlier commit
git tag                      # Lists all the tags
git tag -d v1.0              # Deletes the given tag
```

Branching & Merging

Managing branches

```
git branch bugfix            # Creates a new branch called bugfix
git checkout bugfix          # Switches to the bugfix branch
git switch bugfix            # Same as the above
git switch -C bugfix         # Creates and switches
git branch -d bugfix         # Deletes the bugfix branch
```

Comparing branches

```
git log master..bugfix       # Lists the commits in the bugfix branch not in master
git diff master..bugfix      # Shows the summary of changes
```

Stashing

```
git stash push -m “New tax rules”    # Creates a new stash
git stash list                       # Lists all the stashes
git stash show stash@{1}             # Shows the given stash
git stash show 1                     # shortcut for stash@{1}
git stash apply 1                    # Applies the given stash to the working dir
git stash drop 1                     # Deletes the given stash
git stash clear                      # Deletes all the stashes
```

Merging

```
git merge bugfix             # Merges the bugfix  branch into the current branch
git merge --no-ff bugfix     # Creates a merge commit even if FF is possible
git merge --squash bugfix # Performs a squash merge
git merge --abort            # Aborts the merge
```

Viewing the merged branches

```
git branch --merged       # Shows the merged branches
git branch --no-merged    # Shows the unmerged branches
```

Rebasing

```
git rebase master         # Changes the base of the current branch
```

Cherry picking

```
git cherry-pick dad47ed   # Applies the given commit on the current branch
```

Collaboration

Cloning a repository

```
git clone url
```

Syncing with remotes

```
git fetch origin master         # Fetches master from origin
git fetch origin                # Fetches all objects from origin
git fetch                       # Shortcut for “git fetch origin”
git pull                        # Fetch + merge
git push origin master          # Pushes master to origin
git push                        # Shortcut for “git push origin master”
```

Sharing tags

```
git push origin v1.0            # Pushes tag v1.0 to origin
git push origin —delete v1.0
```

Sharing branches

```
git branch -r                   # Shows remote tracking branches
git branch -vv                  # Shows local & remote tracking branches
git push -u origin bugfix       # Pushes bugfix to origin
git push -d origin bugfix       # Removes bugfix from origin
```

Managing remotes

```
git remote                      # Shows remote repos
git remote add upstream url     # Adds a new remote called upstream
git remote rm upstream          # Remotes upstream
```

Rewriting History

Undoing commits

```
git reset --soft HEAD^        # Removes the last commit, keeps changed staged
git reset --mixed HEAD^       # Unstages the changes as well
git reset --hard HEAD^        # Discards local changes
```

Reverting commits

```
git revert 72856ea            # Reverts the given commit
git revert HEAD~3..           # Reverts the last three commits
git revert --no-commit HEAD~3..
```

Recovering lost commits

```
git reflog                    # Shows the history of HEAD
git reflog show bugfix        # Shows the history of bugfix pointer
```

Amending the last commit

```
git commit --amend
```

Interactive rebasing

```
git rebase -i HEAD~5
```

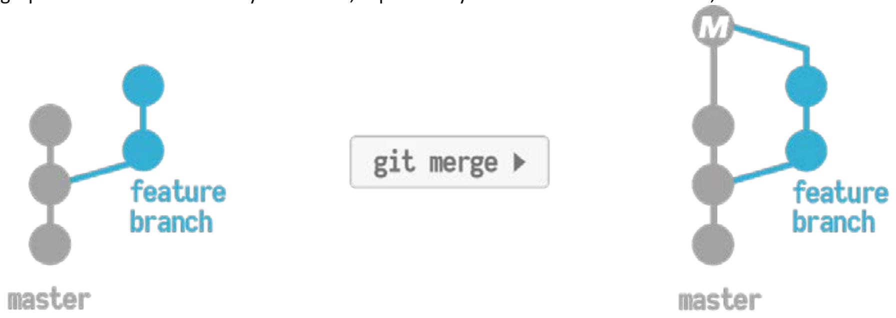
GIT REBASE

The basics

Out of the gate, the goal of both merging and rebasing is to take commits from a feature branch and put them onto another branch. Let's start with how a quote-on-quote "normal" merge makes that happen.

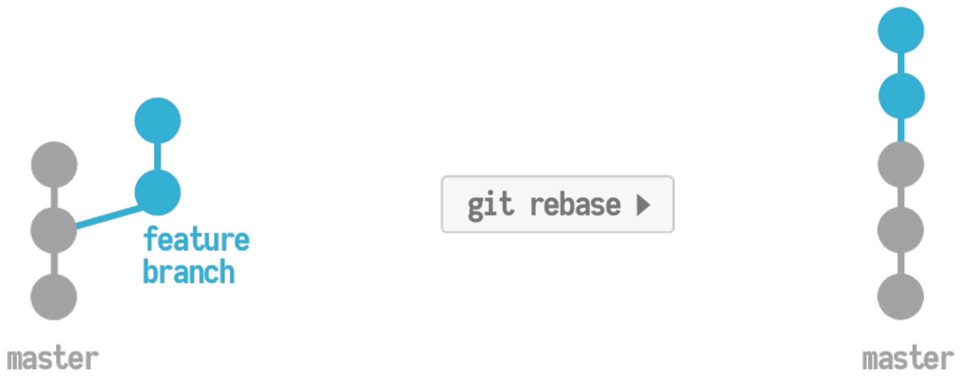
Merging

Say I have a graph that looks like this. As you can see, I split off my feature branch at commit 2, and have done a bit of work.



If I run a *merge*, git will stuff all of my changes from my feature branch into one large *merge* commit that contains *ALL* of my feature branch changes. It will then place this special merge commit onto master. When this happens, the tree will show your feature branch, as well as the master branch. Going further, if you imagine working on a team with other developers, your git tree can become complex: displaying everybody else's branches and merges.

Rebasing



Now let's take a look at how rebase would handle this same situation. Instead of doing a *git merge*, I'll do a *git rebase*. What rebase will do is take all of the commits on your feature branch and move them on top of the master commits. Behind the scenes, git is actually blowing away the feature branch commits and duplicating them as new commits on top of the master branch (remember, under the hood, commit objects are immutable and immovable). What you get with this approach is a nice clean tree with all your commits laid out nicely in a row, like a timeline. Easy to trace.

Rebasing caveats

At this point, I think I better mention some caveats. Rebase doesn't play super well with open-source projects and pull requests since it can be hard to trace, especially small changes that are introduced to a codebase.
It can also be dangerous if you're working on a shared branch with other developers because of how Git rewrites commits when rebasing; however, in the workflow example below, I'll show you how to mitigate this risk.

In practice: the actual commands

When I start development I always make sure the code on my local machine is synced to the latest commit from remote master

```
# With my local master branch checked out
git pull
```

Next, I'll check out a new branch so I can write and commit code to this branch – keeping my work separated from the master branch

```
git checkout -b my_cool_feature
```

As I'm developing my feature, I'll make a few commits…

```
git add .
git commit -m 'This is a new commit, yay!'
```

Note: while I'm developing it's likely that my fellow developers will have shipped some of their own changes to remote master. That's ok, we can deal with that later.

Now that I'm done developing my feature, I want to merge my changes back into remote master. To begin this process I'll switch back to local master branch and pull the latest changes. This ensures my local machine has any new commits submitted by my teammates.

```
git checkout master
git pull
```

What I want to do now is make sure my feature will jive with any new changes from remote master. To do this, I'll checkout my feature branch and rebase against my local master. This will re-anchor my branch against the latest changes I just pulled from remote master. Additionally at this point, Git will let me know if I have any conflicts and I can take care of them on my branch

```
git checkout my_cool_feature
git rebase master
```

Now that my feature branch doesn't have any conflicts, I can switch back to my master branch and place my changes onto master.

```
git checkout master
git rebase my_cool_feature
```

Since I synced with remote master before doing the rebase, I should be able to push my changes up to remote master without issues.

```
git push
```

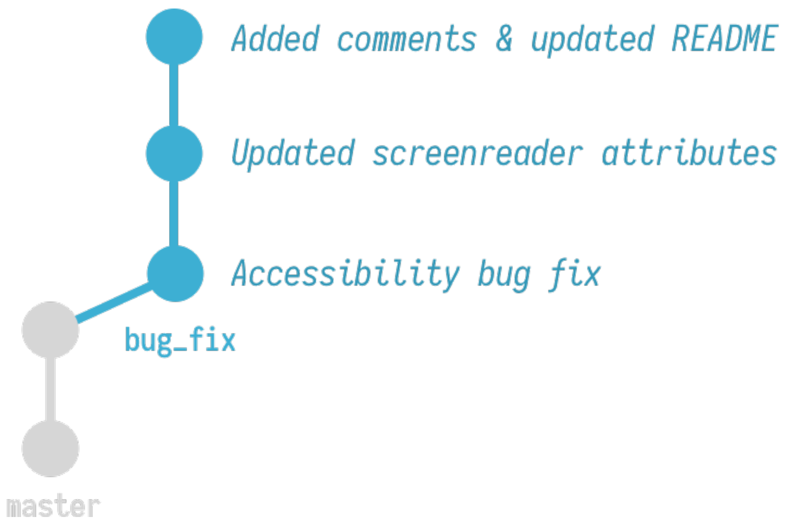
NOTE : This method works for any "parent" branch and its "child" branch and not just master branch and its children branches

GIT SQUASH

Squashing essentially allows you to combine commits together. This is useful for re-packaging commits that are related to eachother in the pursuit of cleaning up history before pushing to an upstream master, or when you're doing some cleanup on your local development branches.

The basics

Say I have a graph that looks like this. You can see I split off some commits onto a bug fix branch:


Added comments & updated README
Updated screenreader attributes
Accessibility bug fix
bug_fix
master

I'll need to get my commits back onto mainline, and I can use merge or rebase to do that; however, with either solution, all my local branch commits would be preserved. Usually, preserving commits like this is a good idea, but let's say you made a typo somewhere and had to create another commit to fix the spelling error. Or you have a bunch of local commits related to a bug fix, but you'd really rather just have all of those related commits under one roof. Enter squashing.

Squashing allows you to rewrite history and combine together commits.

In practice: the actual commands

Now that you know what squash is, let's take a look the actual commands. Again, we'll say my starting point is my bug fix branch with 3 commits.

It would be nice if I didn't have to preserve these extraneous commits as separate entities since they are all related to a bug fix. I'd rather combine them together into one clean commit.

With my bug fix branch checked out, I'll start by running the interactive rebase command with *HEAD~3*. This lets Git know I want to operate on the last three commits back from *HEAD*.

```
git rebase -i HEAD~3
```

Git will open up your default terminal text editor (most likely vim) and present you with a list of commits:

```
pick 7f9d4bf Accessibility fix for frontpage bug

pick 3f8e810 Updated screenreader attributes

pick ec48d74 Added comments & updated README

# Rebase 4095f73..ec48d74 onto 4095f73 (3 commands)

#

# Commands:

# p, pick <commit> = use commit

# r, reword <commit> = use commit, but edit the commit message

# e, edit <commit> = use commit, but stop for amending

# s, squash <commit> = use commit, but meld into previous commit

...
```

There are a couple options here, but we'll go ahead and mark commits we'd like to meld with it's successor by changing *pick* to **squash**. (If you're using VIM, type *i* to enter insert mode)

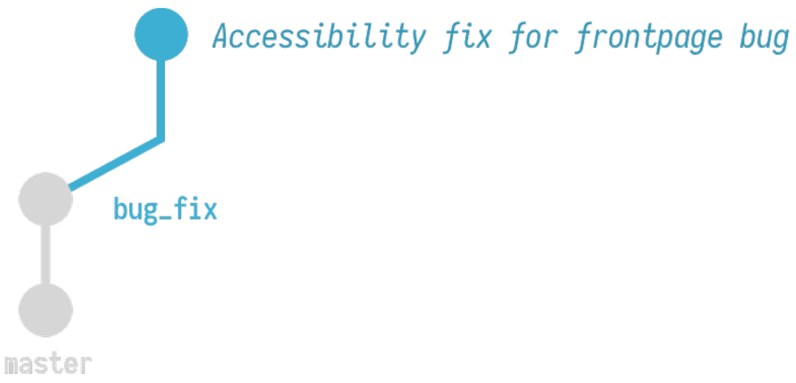
```
pick 7f9d4bf Accessibility fix for frontpage bug

squash 3f8e810 Updated screenreader attributes

squash ec48d74 Added comments & updated README
```

Press *ESC* then type *:wq* to save and exit the file (if you are using VIM)

At this point Git will pop up another dialog where you can rename the commit message of the new, larger squashed commit:

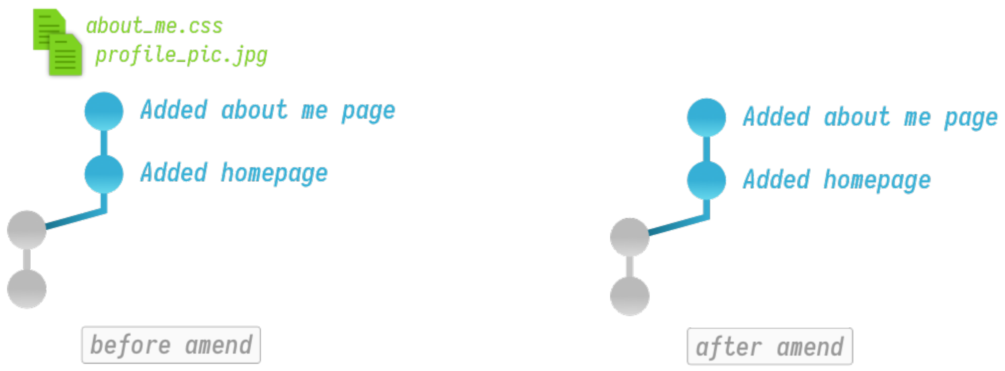
```
# This is a combination of 3 commits

# This is the 1st commit message:

Accessibility fix for frontpage bug

# This is the commit message for #1:

Updated screenreader attributes

# This is the commit message for #2:

Added comments & updated README

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit

...
```

Simply saving this file without making changes will result in a single commit with a commit message that is a concatination of all 3 messages. If you'd rather rename your new commit entirely, comment out each commit's message, and write you're own. Once you've done, save and exit:



That's it. You can either merge or rebase your branch back to mainline.

AMENDING COMMITS



Amending refers to either adding files to, or removing files from a commit. Amend can also be used to change a commit's log message, but today I want to use it to add files to a commit.
As an example: if I run *git status* you can see that I have 2 files that would really like to belong in my latest commit.

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:   about_me.css
        modified:   profile_pic.jpg
```

To get started, I'll first need to stage my files.

```
git add about_me.css profile_pic.jpg
```

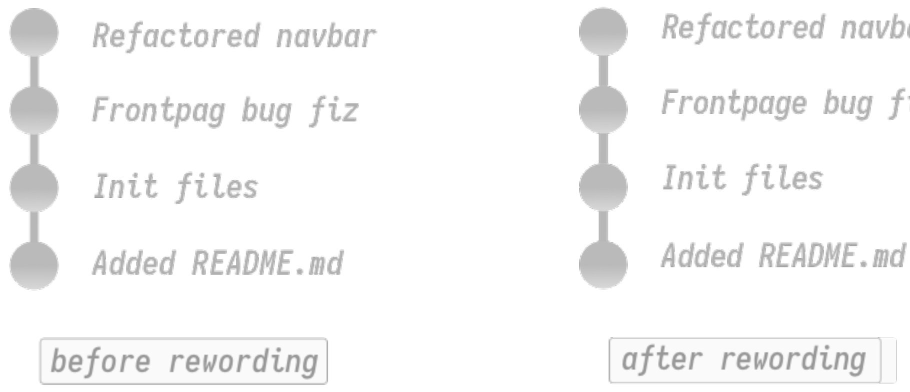
Then I'll use amend to add those staged changes to my most recent commit

```
git commit --amend --no-edit
```

*The **--no-edit** flag tells Git that you'd like to leave the commit message of your latest commit unchanged. If you want to update the commit message, remove this flag.*

That's all there is to it.

REWORDING COMMIT MESSAGES



What if you want to reword a commit message? Enter interactive rebase and its plethora of options. I'll start by telling Git I want to enter interactive rebase and operate upon the last 2 commits back from HEAD.

```
git rebase -i HEAD~2
```

Git will then open up my default terminal text editor (most likely vim) and present me with a file that I'll need to edit.

```
pick 7f9d4bf Frontpag bug fiz
pick 3f8e810 Refactored navbar

# Rebase 4095f73..ec48d74 onto 4095f73 (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
...
```

As you can see, at the top, there is a list of commits, and a big comment section explaining all the different options. Right now I just want to fix that spelling mistake in the first commit. To start, I'll change the prefix of that commit from *pick* to **reword** (If you're using vim, type *i* to enter insert mode).

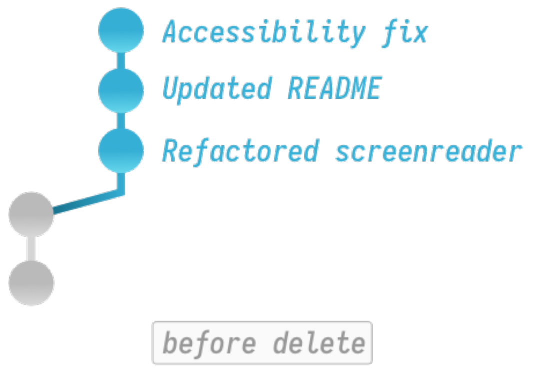
```
reword 7f9d4bf Frontpag bug fiz
pick 3f8e810 Refactored navbar
```

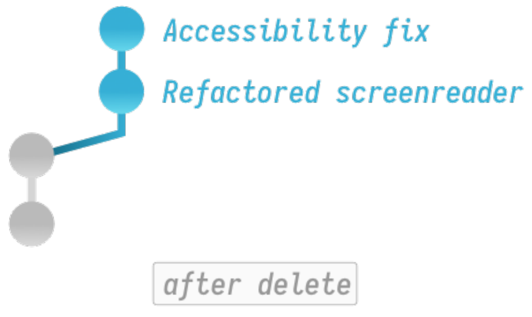
Then I'll save and close the file. (Again, in vim press *ESC* then type *:wq* to save and exit the file). At this point Git will pop up another file where I can rename the commit message. I'll simply edit this file with the reworded commit message.

```
Frontpage bug fix
```

After I'm done making my changes to the commit message, I'll simply save and close the file to finalize the spelling changes.

DELETING COMMITS





As you saw from the rewording example, interactive rebase has many more options, so let's dive into some. Say I'd like to remove a commit from my history.
I'll start by opening interactive rebase again, but this time I'll operate on the last three commits.

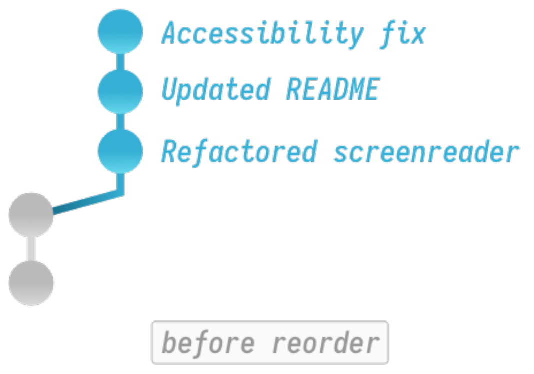
```
git rebase -i HEAD~3
pick 2f8e823 Refactored screenreader attributes
pick 7f9d4bf Updated README
pick 3f8e810 Accessibility fix

# Rebase 4095f73..ec48d74 onto 4095f73 (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
...
```

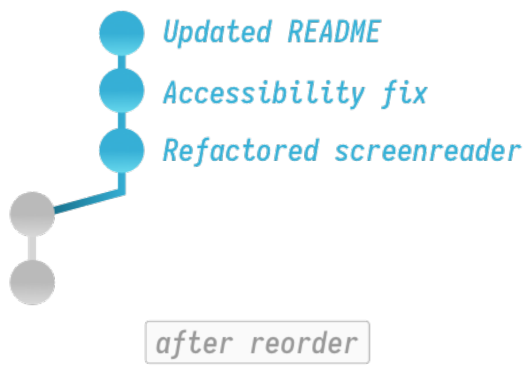
To remove a commit, I simply find the commit I'd like gone and change it's prefix from *pick* to **drop**.

```
pick 2f8e823 Refactored screenreader attributes
drop 7f9d4bf Updated README
pick 3f8e810 Accessibility fix
```

Again, I'll save and quit the file and the commit is gone.

REORDERING COMMITS





Reordering commits is just as easy. We'll enter interactive rebase again

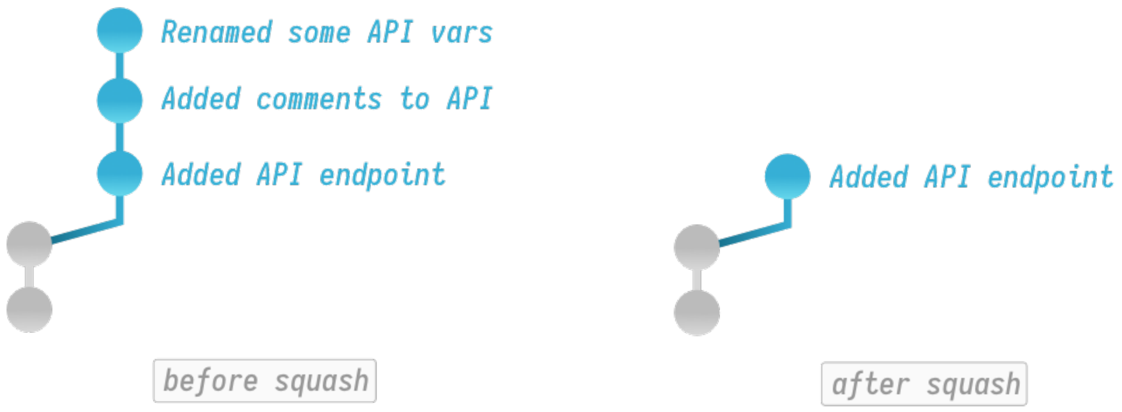
```
git rebase -i HEAD~2
pick 7f9d4bf Updated README
pick 3f8e810 Accessibility fix
# Rebase 4095f73..ec48d74 onto 4095f73 (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
...
```

To reorder, I'll simply change the order of commits by reordering their lines at the top of the file.

```
pick 3f8e810 Accessibility fix
pick 7f9d4bf Updated README
```

Be careful here, if you delete a commit line and save this file without putting it back, Git will destroy the commit entirely.
I'll once again save and quit to finalize the reordering.

SQUASHING COMMITS



Now let's get into some fun stuff. Say I want to meld my previous 2 commits into one. We'll once again use interactive rebase…

```
git rebase -i HEAD~3
pick 7f9d4bf Added API endpoint
pick 3f8e810 Added comments to API
pick ec48d74 Renamed some API vars

# Rebase 4095f73..ec48d74 onto 4095f73 (3 commands)
#
# Commands:
...
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
...
```

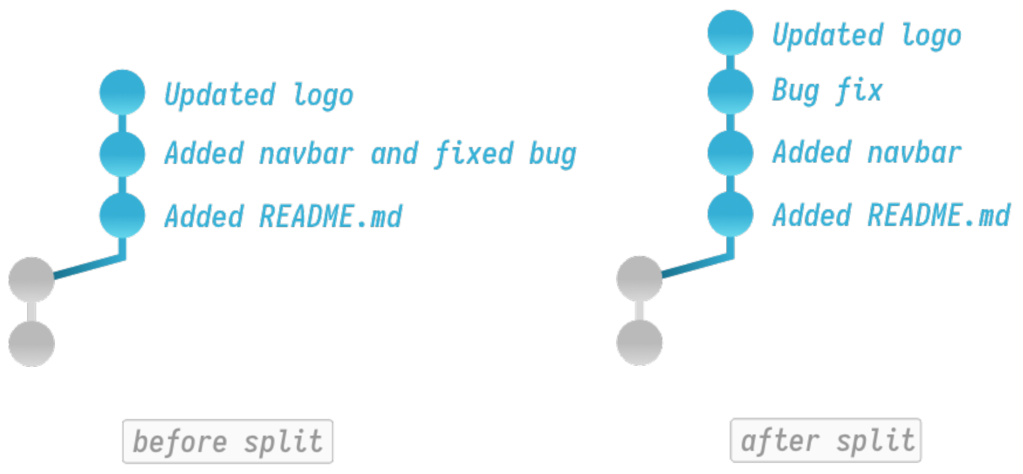
As you can see (above) Git provides two options for combining commits: *squash* and *fixup*. Both options do pretty much the same thing, except squash will kick you to an extra screen which allows you to edit the commit messages. I have a whole other article that goes into greater detail on the squash workflow, so for now I don't care about preserving those other log messages.
I'll change *pick* to **fixup** which indicates to Git that I want to meld this commit into the previous one then discard it's log message.

```
pick 7f9d4bf Added API endpoint
fixup 3f8e810 Added comments to API
fixup ec48d74 Renamed some API vars
```

Now I'll save and quit this file. As you can see, I've ended up with a new squashed commit which contains the contents of all 3 previous commits.

```
git log --oneline

617ebc5 (HEAD -> my_branch) Added API endpoint
```

SPLITTING COMMITS



Lastly, let's talk about splitting commits. Down the line you may decide certain changes would be better off as separate commits and that's what splitting is all about.

```
git log --oneline

3f8e810 Updated logo
ec48d74 Added navbar and fixed bug
2bf910f Added README
```

As you can see above, I'd really prefer if the second commit was split up. To make this happen, I'll enter interactive rebase once again.

```
git rebase -i HEAD~3
pick Added README
pick Added navbar and fixed bug
pick Updated logo

# Rebase 4095f73..ec48d74 onto 4095f73 (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
...
```

As you might guess, the splitting process is a bit more complicated than our previous operations, but don't worry, I'll break it down. First, we need to tell Git which commit we want to change. In this case I want to split the second commit so I'll change it's keyword from *pick* to **edit**

```
pick Added README
edit Added navbar and fixed bug
pick Updated logo
```

Next, I'll save and close the file. When I do this, Git will start executing any rebase operations I specified. Since I left the other commits alone, Git will do nothing to those; however, when it comes to the commit for which I specified *edit*, Git will pause the rebase operation and allow me to make changes.

```
Stopped at f124bc5...  Added navbar and fixed bug
You can amend the commit now, with

  git commit --amend

Once you are satisfied with your changes, run

  git rebase --continue

jacklot in ~/dev/my_site6  (git)-[my_branch|rebase-i]-
>
```

As you can see above, Git dropped me back to the terminal on a special rebasing branch. I can now type commands that will modify our commit. To split this commit, I'll first want to undo the commit by unstaging all the files.

```
git reset HEAD^
```

If I run *git status* you can see I have 3 files that are now unstaged and ready to be re-commited

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)

  analytics.js
  navbar.css
  navbar.html
```

Now I can re-commit the files using the normal add and commit workflow as below

```
git add navbar.css navbar.html
git commit -m 'Added navbar'
git add analytics.js
git commit -m 'Fixed bug'
```

After I'm done re-commiting all my files, I'll just tell Git to continue

```
git rebase --continue
```

That's it. Now if I take a look at my history, you can see I've ended up with 4 commits after splitting

```
git log --oneline

3f8e810 Updated logo
ur48d47 Fixed bug
4c59d82 Added navbar
2bf910f Added README
```

The difference between squash and fixup is that during the rebase, the squash operation will prompt you to combine the messages of the original and the squash commit, whereas the fixup operation will keep the original message and discard the message from the fixup commit.

GitHub CLI

Thodi charcha, thoda gyaan 😎

Resources used :

Official GitHub CLI docs. It’s kinda too wordy and unnecessarily **long and hard** (😏) so this is like a 1hr refresher directly into using most of the useful stuff w/o any probs

Checkout this video just to get refamiliarized with some of the git concepts :

https://www.youtube.com/watch?v=8JJ101D3knE

GitHub CLI

`gh` is GitHub on the command line. It brings pull requests, issues, and other GitHub concepts to the terminal next to where you are already working with git and your code.

Installation

Just head over to https://cli.github.com/ to download the msi file which you can install manually

Authentication

Run `gh auth login` to authenticate with your GitHub account. `gh` will respect tokens set using `GITHUB_TOKEN`.

Setting an editor

To set your preferred editor, you can use `gh config set editor <editor>`. Read more about `gh config`.
Additionally if the above is not set, for macOS and Linux,`gh` will respect the following environment variables, in this order, based on your OS and shell setup:

1. `GIT_EDITOR`
2. `VISUAL`
3. `EDITOR`

On Windows, the editor will currently always be Notepad.

1) HELP (--help)

Option available with –help flag universally can be combined with any cli command

```
      --help   Show help for command
```

2) SHORTCUTS (gh alias set)

Create a shortcut for a gh command

Synopsis

Declare a word as a command alias that will expand to the specified command(s).

The expansion may specify additional arguments and flags. If the expansion includes positional placeholders such as '$1', '$2', etc., any extra arguments that follow the invocation of an alias will be inserted appropriately (**kinda like tab stops**)

This rule only applies for shell scripts (**Windows Powershell**) :

If '–shell' is specified, the alias will be run through a shell interpreter (sh). This allows you to compose commands with "|" or redirect with ">". Note that extra arguments following the alias will not be automatically passed to the expanded expression. To have a shell alias receive arguments, you must explicitly accept them using "$1", "$2", etc., or "$@" to accept all of them.

Platform note: on Windows, shell aliases are executed via "sh" as installed by Git For Windows. If you have installed git on Windows in some other way, shell aliases may not work for you.

Quotes must always be used when defining a command as in the examples.

```
gh alias set <alias> <expansion> [flags]
```

Examples

```
$ gh alias set pv 'pr view'
$ gh pv -w 123
#=> gh pr view -w 123

$ gh alias set bugs 'issue list --label="bugs"'

$ gh alias set epicsBy 'issue list --author="$1" --label="epic"'
$ gh epicsBy vilmibm
#=> gh issue list --author="vilmibm" --label="epic"

$ gh alias set --shell igrep 'gh issue list --label="$1" | grep $2'
$ gh igrep epic foo
#=> gh issue list --label="epic" | grep "foo"
```

Options

```
  -s, --shell   Declare an alias to be passed through a shell interpreter
```

➢ DELETING SHORTCUTS (gh alias delete)

Delete an alias

Synopsis

Delete an alias

```
gh alias delete <alias> [flags]
```

➢ LISTING SHORTCUTS (gh alias list)

List your aliases

Synopsis

This command prints out all of the aliases gh is configured to use.

```
gh alias list [flags]
```

3) CLONE REPO (gh repo clone)

Clone a GitHub repository locally.

If the “OWNER/” portion of the “OWNER/REPO” repository argument is omitted, it defaults to the name of the authenticating user.

Pass additional ‘git clone’ flags by listing them after ‘–’.

```
gh repo clone <repository> [<directory>] [-- <gitflags>...]
```

In use

Using OWNER/REPO syntax

You can clone any repository using OWNER/REPO syntax.

```
# Cloning a repository
~/Projects$ gh repo clone cli/cli
```

Using other selectors

You can also use GitHub URLs to clone repositories.

```
# Cloning a repository
~/Projects/my-project$ gh repo clone https://github.com/cli/cli
```

4) CREATE REPO (gh repo create)

Create a new GitHub repository.

```
gh repo create [<name>] [flags]
```

Examples

```
# create a repository under your account using the current directory name
$ gh repo create

# create a repository with a specific name
$ gh repo create my-project

# create a repository in an organization
$ gh repo create cli/my-project
```

Options

```
  -y, --confirm               Confirm the submission directly
  -d, --description string    Description of repository
      --enable-issues         Enable issues in the new repository (default true)
      --enable-wiki           Enable wiki in the new repository (default true)
  -h, --homepage string       Repository home page URL
      --internal              Make the new repository internal
      --private               Make the new repository private
      --public                Make the new repository public
  -t, --team string           The name of the organization team to be granted
access
  -p, --template string       Make the new repository based on a template
repository
```

In use

With no arguments

Inside a git repository, and with no arguments, `gh` will automatically create a repository on GitHub on your account for your current directory, using the directory name.

```
# Create a repository for the current directory.
~/Projects/my-project$ gh repo create
```

Setting a repository name

Enter a name to set a repository name other than the directory name.

```
# Create a repository in your organization
~/Projects/my-project$ gh repo create my-cool-project
```

Setting your organization as an owner

Use OWNER/REPO syntax to create a repository under an organization that you are a part of.

```
# Create a repository in your organization
~/Projects/my-project$ gh repo create org/repo
```

With flags

Use flags to choose your repository settings.

```
# Create a repository using flags
~/Projects/my-project$ gh repo create --enable-issues=false -public
```

5) FORK REPO (gh repo fork)

Create a fork of a repository.

With no argument, creates a fork of the current repository. Otherwise, forks the specified repository.

```
gh repo fork [<repository>] [flags]
```

Options

```
      --clone    Clone the fork {true|false}
      --remote   Add remote for fork {true|false}
```

In use

With no arguments

Inside a git repository, and without any arguments, we will automatically create a fork on GitHub on your account for your current directory. It will then prompt if you want to set an upstream remote.

```
# Create a fork for the current repository.
```

```
~/Projects/cli$ gh repo fork
```

With arguments

If you pass a repository in OWNER/REPO format, `gh` will automatically create a fork on GitHub on your account and ask if you want to clone it. This works inside or outside of a git repository.

```
# Create a fork for another repository.
~/Projects$ gh repo fork cli/cli
```

Using flags

Use flags to skip prompts about adding a git remote for the fork, or about cloning the forked repository locally.

```
# Skipping remote prompts using flags
~/Projects/cli$ gh repo fork --remote=false
```

6) VIEW REPO (gh repo view)

Synopsis

Display the description and the README of a GitHub repository.

With no argument, the repository for the current directory is displayed.

With '–web', open the repository in a web browser instead.

```
gh repo view [<repository>] [flags]
```

Options

```
  -w, --web    Open a repository in the browser
```

In use

In terminal

By default, we will display items in the terminal.

```
# Viewing a repository in terminal
~/Projects/my-project$ gh repo view owner/repo
```

```
owner/repo
Repository description

  Repository README

View this repository on GitHub: https://github.com/owner/repo/
~/Projects/my-project$
```

In the browser

Quickly open an item in the browser using `--web` or `-w`

```
# Viewing a repository in the browser
~/Projects$ gh repo view owner/repo --web
Opening https://github.com/owner/repo/ in your browser.
~/Projects$
```

With no arguments

Display the repository you're currently in.

```
# Viewing the repository you're in
~/Projects/my-project$ gh repo view
```

```
owner/my-project
Repository description

  Repository README

View this repository on GitHub: https://github.com/owner/repo/
~/Projects/my-project$
```

7) PULL REQUESTS (PR)

➢ CHECKOUT PRs (gh pr checkout)

Check out a pull request in git

Synopsis

Check out a pull request in git

```
gh pr checkout {<number> | <url> | <branch>} [flags]
```

Options

```
      --recurse-submodules   Update all active submodules (recursively)
```

Options inherited from parent commands

```
  -R, --repo OWNER/REPO   Select another repository using the OWNER/REPO format
```

In use

Using pull request number

You can check out any pull request, including from forks, in a repository using its pull request number

```
// Checking out a pull request locally
~/Projects/my-project$ gh pr checkout 12
```

Using other selectors

You can also use URLs and branch names to checkout pull requests.

```
// Checking out a pull request locally
~/Projects/my-project$ gh pr checkout branch-name

~/Projects/my-project$
```

➢ CHECK PR STATUS (gh pr checks)

Show CI status for a single pull request – **mhanje approve kiya ya pending hai ya reject kiya pr ko**

Synopsis

Show CI status for a single pull request

```
gh pr checks [flags]
```

Options inherited from parent commands

```
  -R, --repo OWNER/REPO   Select another repository using the OWNER/RE
```

➢ CLOSING PR (gh pr close)

Close a pull request

Synopsis

Close a pull request

```
gh pr close {<number> | <url> | <branch>} [flags]
```

Options

```
  -d, --delete-branch   Delete the local and remote branch after close
```

Options inherited from parent commands

```
  -R, --repo OWNER/REPO   Select another repository using the OWNER/REPO format
```

➢ CREATING A NEW PR (gh pr create)

Create a pull request

Synopsis

Create a pull request on GitHub.

When the current branch isn't fully pushed to a git remote, a prompt will ask where to push the branch and offer an option to fork the base repository. Use '–head' to explicitly skip any forking or pushing behavior.

A prompt will also ask for the title and the body of the pull request. Use '–title' and '–body' to skip this, or use '–fill' to autofill these values from git commits.

```
gh pr create [flags]
```

Examples

```
$ gh pr create --title "The bug is fixed" --body "Everything works again"
$ gh pr create --reviewer monalisa,hubot
$ gh pr create --project "Roadmap"
$ gh pr create --base develop --head monalisa:feature
```

Options

```
  -a, --assignee login   Assign people by their login
  -B, --base branch      The branch into which you want your code merged
  -b, --body string      Body for the pull request
  -d, --draft            Mark pull request as a draft
  -f, --fill             Do not prompt for title/body and just use commit info
  -H, --head branch      The branch that contains commits for your pull request
(default: current branch)
  -l, --label name       Add labels by name
  -m, --milestone name   Add the pull request to a milestone by name
  -p, --project name     Add the pull request to projects by name
  -r, --reviewer login   Request reviews from people by their login
  -t, --title string     Title for the pull request
  -w, --web              Open the web browser to create a pull request
```

Options inherited from parent commands

```
  -R, --repo OWNER/REPO   Select another repository using the OWNER/REPO format
```

In use

Interactively

```
# Create a pull request interactively
~/Projects/my-project$ gh pr create
```

With flags

```
# Create a pull request using flags
~/Projects/my-project$ gh pr create --title "Pull request title" --body "Pull
request body"
```

In the browser

```
// Quickly navigate to the pull request creation page
~/Projects/my-project$ gh pr create --web
```

Working with forks

This command will automatically create a fork for you if you're in a repository that you don't have permission to push to.

➢ SEE CHANGES TO CODE AS GIVEN IN PR (gh pr diff)

View changes in a pull request

Synopsis

View changes in a pull request

```
gh pr diff [<number> | <url> | <branch>] [flags]
```

Options

```
      --color string   Use color in diff output: {always|never|auto} (default
"auto")
```

Options inherited from parent commands

```
  -R, --repo OWNER/REPO   Select another repository using the OWNER/REPO format
```

➢ LIST ALL PRs IN REPO (gh pr list)

List and filter pull requests in this repository

Synopsis

List and filter pull requests in this repository

```
gh pr list [flags]
```

Examples

```
$ gh pr list --limit 999
$ gh pr list --state closed
$ gh pr list --label "priority 1" --label "bug"
$ gh pr list --web
```

Options

```
  -a, --assignee string   Filter by assignee
  -B, --base string       Filter by base branch
  -l, --label strings     Filter by labels
  -L, --limit int         Maximum number of items to fetch (default 30)
  -s, --state string      Filter by state: {open|closed|merged|all} (default
"open")
  -w, --web               Open the browser to list the pull requests
```

Options inherited from parent commands

```
  -R, --repo OWNER/REPO   Select another repository using the OWNER/REPO format
```

In use

Default behavior

You will see the most recent 30 open items.

```
# Viewing a list of open pull requests
~/Projects/my-project$ gh pr list

Pull requests for owner/repo

#14  Upgrade to Prettier 1.19                           prettier
#14  Extend arrow navigation in lists for MacOS         arrow-nav
#13  Add Support for Windows Automatic Dark Mode        dark-mode
#8   Create and use keyboard shortcut react component   shortcut

~/Projects/my-project$
```

Filtering with flags

You can use flags to filter the list for your specific use cases.

```
# Viewing a list of closed pull requests assigned to a user
~/Projects/my-project$ gh pr list --state closed --assignee user

Pull requests for owner/repo

#13  Upgrade to Electron 7         electron-7
#8   Release Notes Writing Guide   release-notes

~/Projects/my-project$
```

➢ MERGIN PRs (gh pr merge)

Merge a pull request

Synopsis

Merge a pull request on GitHub.

By default, the head branch of the pull request will get deleted on both remote and local repositories. To retain the branch, use '–delete-branch=false'.

```
gh pr merge [<number> | <url> | <branch>] [flags]
```

Options

```
  -d, --delete-branch   Delete the local and remote branch after merge (default
true)
  -m, --merge           Merge the commits with the base branch
  -r, --rebase          Rebase the commits onto the base branch
  -s, --squash          Squash the commits into one commit and merge it into the
base branch
```

Options inherited from parent commands

```
  -R, --repo OWNER/REPO   Select another repository using the OWNER/REPO format
```

➢ REOPEN A CLOSED PR (gh pr reopen)

Reopen a pull request

Synopsis

Reopen a pull request

```
gh pr reopen {<number> | <url> | <branch>} [flags]
```

Options inherited from parent commands

```
  -R, --repo OWNER/REPO   Select another repository using the OWNER/REPO format
```

➢ REVIEWING PRs (gh pr review)

Add a review to a pull request

Synopsis

Add a review to a pull request.

Without an argument, the pull request that belongs to the current branch is reviewed.

```
gh pr review [<number> | <url> | <branch>] [flags]
```

Examples

```
# approve the pull request of the current branch
$ gh pr review --approve

# leave a review comment for the current branch
$ gh pr review --comment -b "interesting"

# add a review for a specific pull request
$ gh pr review 123

# request changes on a specific pull request
$ gh pr review 123 -r -b "needs more ASCII art"
```

Options

```
  -a, --approve           Approve pull request
  -b, --body string       Specify the body of a review
  -c, --comment           Comment on a pull request
  -r, --request-changes   Request changes on a pull request
```

Options inherited from parent commands

```
  -R, --repo OWNER/REPO   Select another repository using the OWNER/REPO format
```

➢ SEE STATUS OF A PR (gh pr status)

Show status of relevant pull requests

Synopsis

Show status of relevant pull requests

```
gh pr status [flags]
```

Options inherited from parent commands

```
  -R, --repo OWNER/REPO   Select another repository using the OWNER/REPO format
```

In use

```
# Viewing the status of your relevant pull requests
~/Projects/my-project$ gh pr status
```

```
Current branch
  #12 Remove the test feature [user:patch-2]
   - All checks failing - Review required
```

```
Created by you
  You have no open pull requests

Requesting a code review from you
  #13 Fix tests [branch]
  - 3/4 checks failing - Review required
  #15 New feature [branch]
   - Checks passing - Approved

~/Projects/my-project$
```

➢ VIEW PRs (gh pr view)

View a pull request

Synopsis

Display the title, body, and other information about a pull request.

Without an argument, the pull request that belongs to the current branch is displayed.

With '–web', open the pull request in a web browser instead.

```
gh pr view [<number> | <url> | <branch>] [flags]
```

Options

```
  -w, --web   Open a pull request in the browser
```

Options inherited from parent commands

```
  -R, --repo OWNER/REPO   Select another repository using the OWNER/REPO format
```

In use

In terminal

By default, we will display items in the terminal.

```
# Viewing a pull request in terminal
~/Projects/my-project$ gh pr view 21
```

In the browser

Quickly open an item in the browser using `--web` or `-w`

```
# Viewing a pull request in the browser
~/Projects/my-project$ gh pr view 21 --web
```

With no arguments

We will display the pull request of the branch you're currently on.

```
# Viewing the pull request of the branch you're on
~/Projects/my-project$ gh pr view
```

8) ISSUES

gh issue

Manage issues

Synopsis

Work with GitHub issues

Examples

```
$ gh issue list
$ gh issue create --label bug
$ gh issue view --web
```

Options

```
  -R, --repo OWNER/REPO   Select another repository using the OWNER/REPO format
```

Options inherited from parent commands

```
      --help   Show help for command
```

Mhanje these 2 options will be available in all commands of gh issue ☝

➢ CLOSING ISSUES (gh issue close)

Close issue

Synopsis

Close issue

```
gh issue close {<number> | <url>} [flags]
```

➢ CREATING AN ISSUE (gh issue create)

Create a new issue

Synopsis

Create a new issue

```
gh issue create [flags]
```

Examples

```
$ gh issue create --title "I found a bug" --body "Nothing works"
$ gh issue create --label "bug,help wanted"
$ gh issue create --label bug --label "help wanted"
$ gh issue create --assignee monalisa,hubot
$ gh issue create --project "Roadmap"
```

Options

```
  -a, --assignee login   Assign people by their login
  -b, --body string      Supply a body. Will prompt for one otherwise.
  -l, --label name       Add labels by name
  -m, --milestone name   Add the issue to a milestone by name
  -p, --project name     Add the issue to projects by name
  -t, --title string     Supply a title. Will prompt for one otherwise.
  -w, --web              Open the browser to create an issue
OWNER/REPO format
```

In use

Interactively

```
# Create an issue interactively
~/Projects/my-project$ gh issue create
```

With flags

```
# Create an issue using flags
~/Projects/my-project$ gh issue create --title "Issue title" --body "Issue body"
```

In the browser

```
// Quickly navigate to the issue creation page
~/Projects/my-project$ gh issue create --web
```

➢ LISTING ISSUES (gh issue list)

List and filter issues in this repository

Synopsis

List and filter issues in this repository

```
gh issue list [flags]
```

Examples

```
$ gh issue list -l "help wanted"
$ gh issue list -A monalisa
$ gh issue list --web
$ gh issue list --milestone 'MVP'
```

Options

```
  -a, --assignee string    Filter by assignee
  -A, --author string      Filter by author
  -l, --label strings      Filter by labels
  -L, --limit int          Maximum number of issues to fetch (default 30)
      --mention string     Filter by mention
  -m, --milestone number   Filter by milestone number or `title`
  -s, --state string       Filter by state: {open|closed|all} (default "open")
  -w, --web                Open the browser to list the issue(s)
using the OWNER/REPO format
```

In use

Default behavior

You will see the most recent 30 open items.

```
# Viewing a list of open issues
~/Projects/my-project$ gh issue list
```

```
Issues for owner/repo

#14  Update the remote url if it changed  (bug)
#14  PR commands on a detached head       (enhancement)
#13  Support for GitHub Enterprise        (wontfix)
#8   Add an easier upgrade command        (bug)
```

Filtering with flags

You can use flags to filter the list for your specific use cases.

```
# Viewing a list of closed issues assigned to a user
~/Projects/my-project$ gh issue list --state closed --assignee user
```

➢ REOPEN CLOSED ISSUES (gh issue reopen)

Reopen issue

Synopsis

Reopen issue

```
gh issue reopen {<number> | <url>} [flags]
using the OWNER/REPO format
```

➢ VIEWIN ISSUE STATUS (gh issue status)

Show status of relevant issues

Synopsis

Show status of relevant issues

```
gh issue status [flags]using the OWNER/REPO format
```

In use

```
# Viewing issues relevant to you
~/Projects/my-project$ gh issue status
```

```
Issues assigned to you
  #8509 [Fork] Improve how Desktop handles forks  (epic:fork, meta)

Issues mentioning you
  #8938 [Fork] Add create fork flow entry point at commit warning  (epic:fork)
  #8509 [Fork] Improve how Desktop handles forks  (epic:fork, meta)

Issues opened by you
  #8936 [Fork] Hide PR number badges on branches that have an upstream PR
(epic:fork)
  #6386 Improve no editor detected state on conflicts modal  (enhancement)
```

➢ VIEWIN ISSUES (gh issue view)

View an issue

Synopsis

Display the title, body, and other information about an issue.

With '–web', open the issue in a web browser instead.

```
gh issue view {<number> | <url>} [flags]
```

Options

```
  -w, --web   Open an issue in the browser
repository using the OWNER/REPO format
```

In use

In terminal

By default, we will display items in the terminal.

```
# Viewing an issue in terminal
~/Projects/my-project$ gh issue view 21
```

```
Issue title
opened by user. 0 comments. (label)

  Issue body

View this issue on GitHub: https://github.com/owner/repo/issues/21
~/Projects/my-project$
```

In the browser

Quickly open an item in the browser using `--web` or `-w`

```
# Viewing an issue in the browser
~/Projects/my-project$ gh issue view 21 --web
```