

git init → create brand new repo on computer

git clone → copy existing repos from somewhere else to your PC

git status → check status

git log → shows existing commits

--oneline → compact version

--stat → displays file changes

--patch or -p → shows the actual changes into the files

-w → whitespaces are ignored

--decorate → shows additional info

```
$ git log -p fdf5493
```

By supplying a SHA, the `git log -p` command will start at that commit! No need to scroll through everything! Keep in mind that it will *also* show all of the commits that were made *prior* to the supplied SHA.

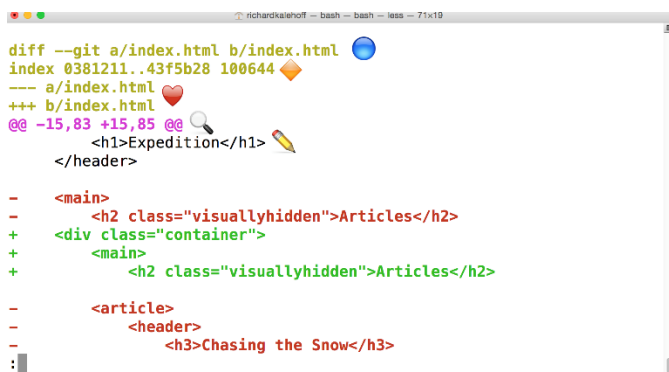
git show → displays info about the given commit

add SHA → it will show info about just that commit

Navigating The Log

If you're not used to a pager on the command line, navigating in [Less](#) can be a bit odd. Here are some helpful keys:

- to scroll **down**, press
 - `j` or `↓` to move *down* one line at a time
 - `d` to move by half the page screen
 - `f` to move by a whole page screen
- to scroll **up**, press
 - `k` or `↑` to move *up* one line at a time
 - `u` to move by half the page screen
 - `b` to move by a whole page screen
- press `q` to **quit** out of the log (returns to the regular command prompt)








```
diff --git a/index.html b/index.html
index 0381211..43f5b28 100644
--- a/index.html
+++ b/index.html
@@ -15,83 +15,85 @@
<h1>Expedition</h1>
</header>

- <main>
-   <h2 class="visuallyhidden">Articles</h2>
+ <div class="container">
+   <main>
+     <h2 class="visuallyhidden">Articles</h2>
+
+     <article>
+       <header>
+         <h3>Chasing the Snow</h3>
```

Annotated `git log -p` Output

Using the image above, let's do a quick recap of the `git log -p` output:

-  - the file that is being displayed
-  - the hash of the first version of the file and the hash of the second version of the file
 - not usually important, so it's safe to ignore
-  - the old version and current version of the file
-  - the lines where the file is added and how many lines there are
 - `-15,83` indicates that the old version (represented by the `-`) started at line 15 and that the file had 83 lines
 - `+15,85` indicates that the current version (represented by the `+`) starts at line 15 and that there are now 85 lines...these 85 lines are shown in the patch below
-  - the actual changes made in the commit
 - lines that are red and start with a minus (`-`) were in the original version of the file but have been removed by the commit
 - lines that are green and start with a plus (`+`) are new lines that have been added in the commit

git add → add files from the working directory to the staging index

. → in order to add everything

git commit → take files from staging index and save them in the repository

if vim editor: fill first line with commit comment -> ESC -> :x

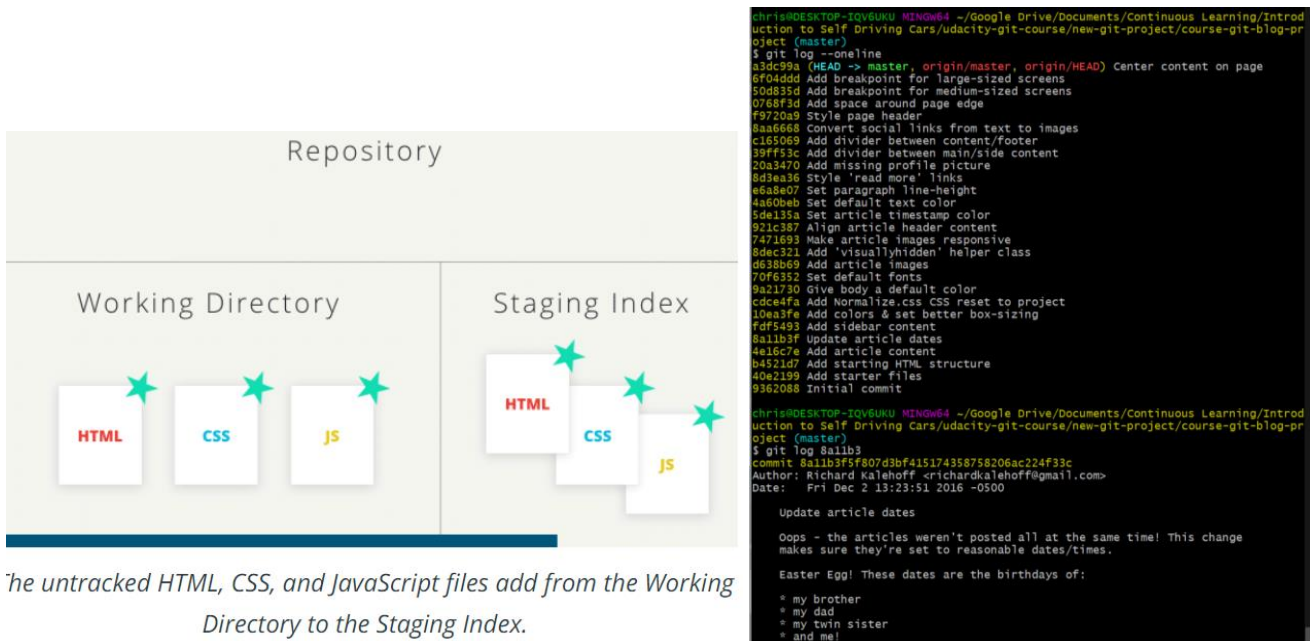
-m "comment" -> bypasses the editor

-a -m "comment" -> to remove also deleted files in remote repo

Comments should say what the change is.

If you need to explain why, add an empty line in the vim editor and then the explanatory text

git diff → see changes that have been made but haven't been committed, yet



Git Ignore

If you want to keep a file in your project's directory structure but make sure it isn't accidentally committed to the project, you can use the specially named file, `.gitignore` (note the dot at the front, it's important!). Add this file to your project in the same directory that the hidden `.git` directory is located. All you have to do is list the *names* of files that you want Git to ignore (not track) and it will ignore them.

Let's try it with the "project.docx" file. Add the following line inside the `.gitignore` file:

```
project.docx
```

Now run `git status` and check its output:

```
richardkalehoff (master) new-git-project
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
nothing added to commit but untracked files present (use "git add" to track)
richardkalehoff (master) new-git-project
$
```

QUESTION 1 OF 2

Which of the following files will be ignored if `*.png` is entered into the `.gitignore` file?

- ☐ ocean.jpg
- ☒ trees.png
- ☐ png-format.pdf
- ☐ not-a-png.jpeg
- ☒ bg-pattern.png
- ☐ logo.gif
- ☒ LOUDFILE.PNG

SUBMIT

QUESTION 2 OF 2

If you ask Git to ignore "be?rs", which of the following filenames will be ignored?

- ☒ bears
- ☐ beavers
- ☐ BeAr5
- ☒ beers
- ☐ boars

Globbering Crash Course

Let's say that you add 50 images to your project, but want Git to ignore all of them. Does this mean you have to list each and every filename in the `.gitignore` file? Oh gosh no, that would be crazy! Instead, you can use a concept called [globbing](#).

Globbering lets you use special characters to match patterns/characters. In the `.gitignore` file, you can use the following:

- blank lines can be used for spacing
- `#` - marks line as a comment
- `*` - matches 0 or more characters
- `?` - matches 1 character
- `[abc]` - matches a, b, _or_ c
- `**` - matches nested directories - `a/**/z` matches
 - `a/z`
 - `a/b/z`
 - `a/b/c/z`

`git tag` → give tags to specific commits

-a "name of the tag" → to indicate what tag name is

SHA → indicate such tag in such commit

--delete or -d "name of the tag" → to remove such tag

QUESTION 1 OF 3

By default, a Git tag will not appear in a log. What flag must be used to display the tag information in the output of `git log`?

- ☐ --show-tags
- ☐ --tags
- ☐ --display-all
- ☒ --decorate



`git branch` → create different lines of development

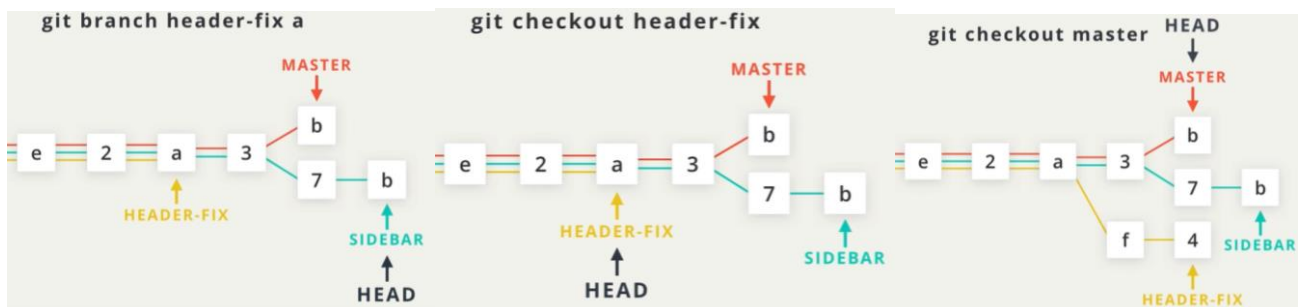
"name of the branch" → create branch

-d "name of the branch" → delete such branch

Git won't let you delete a branch if it has commits on it that aren't on any other branch

Git wouldn't let you delete the branch because you can't delete a branch that you're currently on

-D "name of the branch" → force removal



QUESTION 2 OF 3

From what you know about both the `git branch` and `git tag` commands, what do you think the following command will do?

```
$ git branch alt-sidebar-loc 42a69f
```

- ☐ will create a branch `alt` at the same commit as the `master` branch
- ☐ will create the 3 branches `alt`, `sidebar`, `loc`
- ☐ will move the master branch to the commit with SHA `42a69f`
- ☒ will create the `alt-sidebar-loc` branch and have it point to the commit with SHA `42a69f`

`git checkout` → switch between different branches and tags

“name of branch” → change header to that branch

💡 Switch and Create Branch In One Command 💡

The way we currently work with branches is to create a branch with the `git branch` command and then switch to that newly created branch with the `git checkout` command.

But did you know that the `git checkout` command can actually create a new branch, too? If you provide the `-b` flag, you can create a branch and switch to it all in one command.

```
$ git checkout -b richards-branch-for-awesome-changes
```

It's a pretty useful command, and I use it often.

See All Branches At Once

We've made it to the end of all the changes we needed to make! Awesome job!

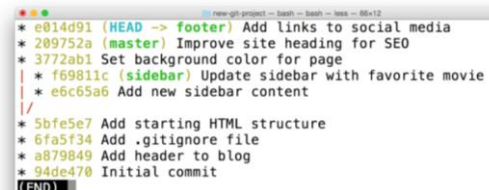
Now we have multiple sets of changes on three different branches. We can't see other branches in the `git log` output unless we switch to a branch. Wouldn't it be nice if we could see *all* branches at once in the `git log` output.

As you've hopefully learned by now, the `git log` command is pretty powerful and *can* show us this information. We'll use the new `--graph` and `--all` flags:

```
$ git log --oneline --decorate --graph --all
```

The `--graph` flag adds the bullets and lines to the leftmost part of the output. This shows the actual *branching* that's happening. The `--all` flag is what displays *all* of the branches in the repository.

Running this command will show all branches and commits in the repository:



```
main@main:~/projects/learn-git$ git log --oneline --decorate --graph --all
* e014d91 (HEAD -> footer) Add links to social media
* 209752a (master) Improve site heading for SEO
* 3772ab1 Set background color for page
| * f69811c (sidebar) Update sidebar with favorite movie
| * e6c65a6 Add new sidebar content
|/
* 5bfe5e7 Add starting HTML structure
* 6fa5f34 Add .gitignore file
* a879849 Add header to blog
* 94de470 Initial commit
(END)
```

`git merge` → combine changes on different branches

When we merge, we're merging some other branch into the current (checked-out) branch

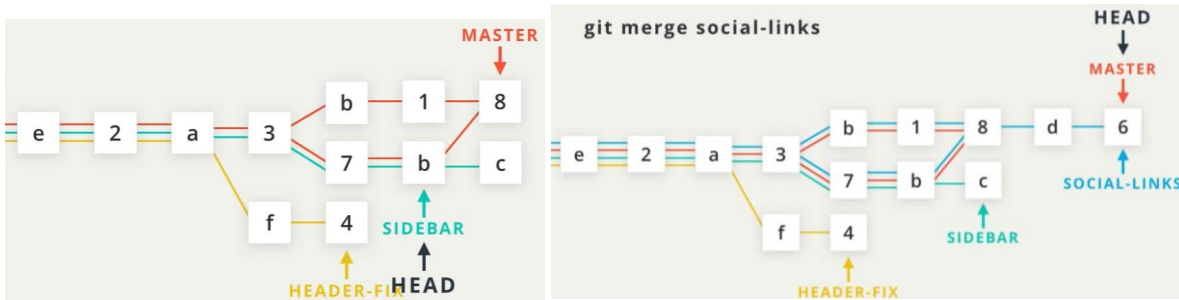
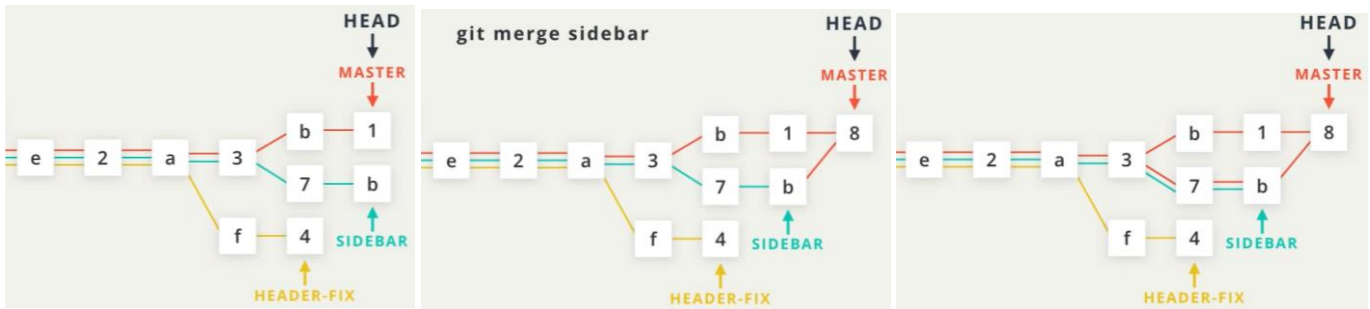
⚠️ Know The Branch ⚠️

It's very important to know which branch you're on when you're about to merge branches together. Remember that making a merge makes a commit.

As of right now, we do not know how to undo changes. We'll go over it in the next lesson, but if you make a merge on the wrong branch, use this command to undo the merge:

```
git reset --hard HEAD^
```

(Make sure to include the `^` character! It's known as a "Relative Commit Reference" and indicates "the parent commit". We'll look at Relative Commit References in the next lesson.)



```
* e014d91 (HEAD -> master, footer) Add links to social media
* 209752a Improve site heading for SEO
* 3772ab1 Set background color for page
* 5bfe5e7 Add starting HTML structure
* 6fa5f34 Add .gitignore file
* a879849 Add header to blog
* 94de470 Initial commit
(END)
```

The Terminal application showing the result of merging the `footer` branch into the `master` branch.

Merge Conflict Indicators Explanation

The editor has the following merge conflict indicators:

- <<<<<< HEAD everything below this line (until the next indicator) shows you what's on the current branch
- | | | | | merged common ancestors everything below this line (until the next indicator) shows you what the original lines were
- ===== is the end of the original lines, everything that follows (until the next indicator) is what's on the branch that's being merged in
- >>>>>> heading-update is the ending indicator of what's on the branch that's being merged in (in this case, the `heading-update` branch)

Resolving A Merge Conflict

Git is using the merge conflict indicators to show you what lines caused the merge conflict on the two different branches as well as what the original line used to have. So to resolve a merge conflict, you need to:

1. choose which line(s) to keep
2. remove all lines with indicators

For some reason, I'm not happy with the word "Crusade" right now, but "Quest" isn't all that exciting either. How about "Adventurous Quest" as a heading?!

`git commit --amend` → alter the most-recent commit

Changing The Last Commit

You've already made plenty of commits with the `git commit` command. Now with the `--amend` flag, you can alter the *most-recent* commit.

```
$ git commit --amend
```

If your Working Directory is clean (meaning there aren't any uncommitted changes in the repository), then running `git commit --amend` will let you provide a new commit message. Your code editor will open up and display the original commit message. Just fix a misspelling or completely reword it! Then save it and close the editor to lock in the new commit message.

Add Forgotten Files To Commit

Alternatively, `git commit --amend` will let you include files (or changes to files) you might've forgotten to include. Let's say you've updated the color of all navigation links across your entire website. You committed that change and thought you were done. But then you discovered that a special nav link buried deep on a page doesn't have the new color. You *could* just make a new commit that updates the color for that one link, but that would have two back-to-back commits that do practically the exact same thing (change link colors).

Instead, you can amend the last commit (the one that updated the color of all of the other links) to include this forgotten one. To do get the forgotten link included, just:

- edit the file(s)
- save the file(s)
- stage the file(s)
- and run `git commit --amend`

So you'd make changes to the necessary CSS and/or HTML files to get the forgotten link styled correctly, then you'd save all of the files that were modified, then you'd use `git add` to stage all of the modified files (just as if you were going to make a new commit!), but then you'd run `git commit --amend` to update the most-recent commit instead of creating a new one.

`git revert` → reverses given commit

Revert Recap

To recap, the `git revert` command is used to reverse a previously made commit:

```
$ git revert <SHA-of-commit-to-revert>
```

This command:

- will undo the changes that were made by the provided commit
- creates a new commit to record the change

`git reset` → delete commits (only possible to do it sequentially) [POTENTIALLY DANGEROUS!]

--mixed
--soft
--hard

Relative Commit References

You already know that you can reference commits by their SHA, by tags, branches, and the special `HEAD` pointer. Sometimes that's not enough, though. There will be times when you'll want to reference a commit relative to another commit. For example, there will be times where you'll want to tell Git about the commit that's one before the current commit...or two before the current commit. There are special characters called "Ancestry References" that we can use to tell Git about these relative references. Those characters are:

- `^` - indicates the parent commit
- `~` - indicates the *first* parent commit

Here's how we can refer to previous commits:

- the parent commit - the following indicate the parent commit of the current commit
 - `HEAD^`
 - `HEAD~`
 - `HEAD~1`
- the grandparent commit - the following indicate the grandparent commit of the current commit
 - `HEAD^^`
 - `HEAD~2`
- the great-grandparent commit - the following indicate the great-grandparent commit of the current commit
 - `HEAD^^^`
 - `HEAD~3`

The main difference between the `^` and the `~` is when a commit is created *from a merge*. A merge commit has *two* parents. With a merge commit, the `^` reference is used to indicate the *first* parent of the commit while `^2` indicates the *second* parent. The first parent is the branch you were on when you ran `git merge` while the second parent is the branch that was merged in.

```
* 9ec85ca (HEAD -> master) Revert "Set page heading to "Quests & Crusades""
* db7e87a Set page heading to "Quests & Crusades"
* 796ddb0 Merge branch 'heading-update'
|\
| * 4c9749e (heading-update) Set page heading to "Crusade"
* | 0c5975a Set page heading to "Quest"
|/
* 1a56a81 Merge branch 'sidebar'
|\
| * f69811c (sidebar) Update sidebar with favorite movie
| * e6c65a6 Add new sidebar content
* | e014d91 (footer) Add links to social media
* | 209752a Improve site heading for SEO
* | 3772ab1 Set background color for page
|/
* 5bfe5e7 Add starting HTML structure
* 6fa5f34 Add .gitignore file
* a879849 Add header to blog
* 94de470 Initial commit
```

Let's look at how we'd refer to some of the previous commits. Since `HEAD` points to the `9ec85ca` commit:

- `HEAD^` is the `db7e87a` commit
- `HEAD~1` is also the `db7e87a` commit
- `HEAD^^` is the `796ddb0` commit
- `HEAD~2` is also the `796ddb0` commit
- `HEAD^^^` is the `0c5975a` commit
- `HEAD~3` is also the `0c5975a` commit
- `HEAD^^^2` is the `4c9749e` commit (this is the grandparent's (`HEAD^^`) second parent (`^2`))

QUESTION 1 OF 4

Which commit is referenced by `HEAD~6`?

- ☐ 4c9749e
- ☐ 0c5975a
- ☐ 1a56a81
- ☐ f69811c
- ☐ e014d91

☒ 209752a

SUBMIT

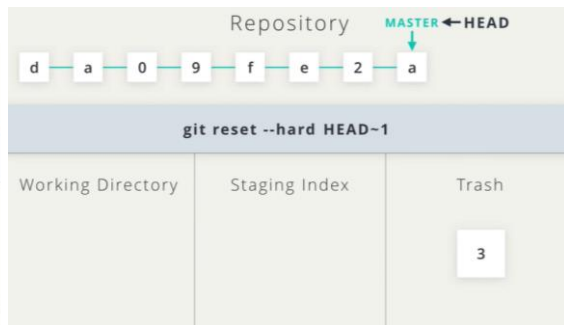
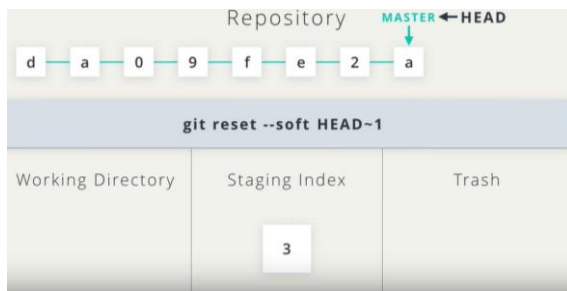
You did so well on that last one, why not give this one a go! Using the same repository, which commit is referenced by `HEAD~4^2`?

f69811c

RESET

By default flag is set to mixed:





Reset's `--mixed` Flag

Let's look at each one of these flags.

```
* 9ec05ca (HEAD -> master) Revert "Set page heading to "Quests & Crusades""
* db7e87a Set page heading to "Quests & Crusades"
* 796ddb0 Merge branch 'heading-update'
```

Using the sample repo above with `HEAD` pointing to `master` on commit `9ec05ca`, running `git reset --mixed HEAD^` will take the changes made in commit `9ec05ca` and move them to the working directory.

```
richardkalehoff (master) new-git-project
$ git reset --mixed HEAD^
Unstaged changes after reset:
M   index.html
richardkalehoff (master *) new-git-project
$ git status
On branch master
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:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
richardkalehoff (master *) new-git-project
$
```

💡 Back To Normal 💡

If you created the `backup` branch prior to resetting anything, then you can easily get back to having the `master` branch point to the same commit as the `backup` branch. You'll just need to:

1. remove the uncommitted changes from the working directory
2. merge `backup` into `master` (which will cause a Fast-forward merge and move `master` up to the same point as `backup`)

```
$ git checkout -- index.html
$ git merge backup
```

`git remote` → interface for managing remote connections you have to other repositories

`-v` → includes URL

`add <name> <URL>`

`rename <old name> <new name>`

`rm <name>` → to remove the connection

git fetch → downloads commits, files and refs from a remote repository (like update) of all branches

<branch> → only fetches a branch

--all → fetches from all registered remotes

--dry-run → a simulation of a fetch

git fetch vs git pull → You can consider git fetch the 'safe' version of the two commands. It will download the remote content but not update your local repo's working state, leaving your current work intact. git pull is the more aggressive alternative, it will download the remote content for the active local branch and immediately execute git merge to create a merge commit for the new remote content. If you have pending changes in progress this will cause conflicts and kickoff the merge conflict resolution flow.

git push → upload your local repository to a remote repository (counterpart of fetch). To prevent you from overwriting commits, Git won't let you push when it results in a non-fast-forward merge in the destination repository

<remote>

--force → force push even if it results in a non-fast-forward merge

--all → push all of your local branches to the specified remote

--tags → sends all of your local tags to the remote repo since they are not automatically uploaded

Amended force push

The `git commit` command accepts a `--amend` option which will update the previous commit. A commit is often amended to update the commit message or add new changes. Once a commit is amended a `git push` will fail because Git will see the amended commit and the remote commit as diverged content. The `--force` option must be used to push an amended commit.

```
# make changes to a repo and git add
git commit --amend
# update the existing commit message
git push --force origin master
```

The above example assumes it is being executed on an existing repository with a commit history. `git commit --amend` is used to update the previous commit. The amended commit is then force pushed using the `--force` option.

Deleting a remote branch or tag

Sometimes branches need to be cleaned up for book keeping or organizational purposes. To fully delete a branch, it must be deleted locally and also remotely.

```
git branch -D branch_name
git push origin :branch_name
```

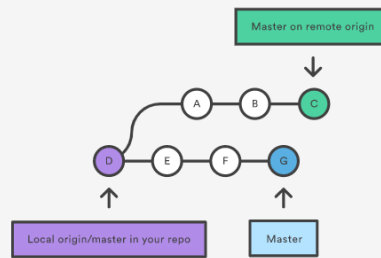
The above will delete the remote branch named `branch_name` passing a branch name prefixed with a colon to `git push` will delete the remote branch.

git pull → used to fetch and download content from a remote repo and immediately update the local repo to match the content [fetch + merge]

<remote>

--rebase

The `git pull` command first runs `git fetch` which downloads content from the specified remote repository. Then a `git merge` is executed to merge the remote content refs and heads into a new local merge commit. To better demonstrate the pull and merging process let us consider the following example. Assume we have a repository with a master branch and a remote origin.



In this scenario, `git pull` will download all the changes from the point where the local and master diverged. In this example, that point is E. `git pull` will fetch the diverged remote commits which are A-B-C. The pull process will then create a new local merge commit containing the content of the new diverged remote commits.

