fournova

GIT CHEAT SHEET

presented by **TOWER** › Version control with Git - made easy

CREATE

Clone an existing repository
```
$ git clone ssh://user@domain.com/repo.git
```

Create a new local repository
```
$ git init
```

LOCAL CHANGES

Changed files in your working directory
```
$ git status
```

Changes to tracked files
```
$ git diff
```

Add all current changes to the next commit
```
$ git add .
```

Add some changes in <file> to the next commit
```
$ git add -p <file>
```

Commit all local changes in tracked files
```
$ git commit -a
```

Commit previously staged changes
```
$ git commit
```

Change the last commit
Don't amend published commits!
```
$ git commit --amend
```

COMMIT HISTORY

Show all commits, starting with newest
```
$ git log
```

Show changes over time for a specific file
```
$ git log -p <file>
```

Who changed what and when in <file>
```
$ git blame <file>
```

BRANCHES & TAGS

List all existing branches
```
$ git branch -av
```

Switch HEAD branch
```
$ git checkout <branch>
```

Create a new branch based on your current HEAD
```
$ git branch <new-branch>
```

Create a new tracking branch based on a remote branch
```
$ git checkout --track <remote/branch>
```

Delete a local branch
```
$ git branch -d <branch>
```

Mark the current commit with a tag
```
$ git tag <tag-name>
```

UPDATE & PUBLISH

List all currently configured remotes
```
$ git remote -v
```

Show information about a remote
```
$ git remote show <remote>
```

Add new remote repository, named <remote>
```
$ git remote add <shortname> <url>
```

Download all changes from <remote>, but don't integrate into HEAD
```
$ git fetch <remote>
```

Download changes and directly merge/integrate into HEAD
```
$ git pull <remote> <branch>
```

Publish local changes on a remote
```
$ git push <remote> <branch>
```

Delete a branch on the remote
```
$ git branch -dr <remote/branch>
```

Publish your tags
```
$ git push --tags
```

MERGE & REBASE

Merge <branch> into your current HEAD
```
$ git merge <branch>
```

Rebase your current HEAD onto <branch>
Don't rebase published commits!
```
$ git rebase <branch>
```

Abort a rebase
```
$ git rebase --abort
```

Continue a rebase after resolving conflicts
```
$ git rebase --continue
```

Use your configured merge tool to solve conflicts
```
$ git mergetool
```

Use your editor to manually solve conflicts and (after resolving) mark file as resolved
```
$ git add <resolved-file>
$ git rm <resolved-file>
```

UNDO

Discard all local changes in your working directory
```
$ git reset --hard HEAD
```

Discard local changes in a specific file
```
$ git checkout HEAD <file>
```

Revert a commit (by producing a new commit with contrary changes)
```
$ git revert <commit>
```

Reset your HEAD pointer to a previous commit

...and discard all changes since then
```
$ git reset --hard <commit>
```

...and preserve all changes as unstaged changes
```
$ git reset <commit>
```

...and preserve uncommitted local changes
```
$ git reset --keep <commit>
```

git cheat sheet

learn more about git the simple way at *rogerdudler.github.com/git-guide/*
cheat sheet created by Nina Jaeschke of *ninagrafik.com*

create & clone

create new repository	*git init*
clone local repository	*git clone /path/to/repository*
clone remote repository	*git clone username@host:/path/to/repository*

add & remove

add changes to INDEX	*git add <filename>*
add all changes to INDEX	*git add **
remove/delete	*git rm <filename>*

commit & synchronize

commit changes	*git commit -m "Commit message"*
push changes to remote repository	*git push origin master*
connect local repository to remote repository	*git remote add origin <server>*
update local repository with remote changes	*git pull*

branches

create new branch	*git checkout -b <branch>* *e.g. git checkout -b feature_x*
switch to master branch	*git checkout master*
delete branch	*git branch -d <branch>*
push branch to remote repository	*git push origin <branch>*

merge

merge changes from another branch	*git merge <branch>*
view changes between two branches	*git diff <source_branch> <target_branch>* *e.g. git diff feature_x feature_y*

tagging

create tag	*git tag <tag> <commit ID>* *e.g. git tag 1.0.0 1b2e1d63ff*
get commit IDs	*git log*

restore

replace working copy with latest from HEAD	*git checkout -- <filename>*

-- SETUP GIT REPOSITORY

- Cloning a Git repository:

```
git clone https://github.com/miyagawa/cpanminus.git
```

- Create a Git repository (with the hidden files of Git to track our file changes) from a local folder:

```
git init
```

- Add the cloud repository Git URL to Git (to refer it later by the name of “origin")

```
git remote add origin <CLOUD_REPOSITORY_CLONE_URL>
```

- Check Git status (to check the current branch and all the files modified)

```
git status
```

- Create a new branch and switch to it

```
git checkout -b <NEW_BRANCH_NAME>
```

- Switch to existing branch

```
git checkout <EXISTING_BRANCH_NAME>
```

- Add files to the staging area for a commit

```
git add .
```

- Create a Git commit

```
git commit -m <YOUR_COMMIT_MESSAGE>
```

- Push commits in your branch to cloud repository with a specified branch name

```
git push origin <YOUR_BRANCH_NAME>
```

-- MISC

- Retrieve a single file from a repository

```
git archive --remote=ssh://host/pathto/repo.git HEAD README.md
```

- Retrieve a single file from another branch

```
# first get back to master
git checkout master

# then copy the version of app.js from branch "experiment"
git checkout experiment -- app.js (-- indicates given files)
```

-- FIXING MISTAKES WITH GIT

- Show a list of all changes that haven't been committed/not staged for commit (along with branch name)

If you didn't commit the code, `git status` is a helpful command. It shows a list of all changes that haven't been committed yet, and also helpfully tells you what to do to ignore some of the uncommitted changes. The output has these sections:

```
On branch feature-user-account
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
...
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)
```

If the changes you want to ignore are in the first part, the suggested command

```
git reset HEAD
```

(`HEAD` = commit identifier)

will bring them to the second group. And for changes in the second group, the suggested command

```
git checkout --
```

(the `--` is important) will remove all changes from the given files. So make sure all the changes in those files are something you want to get rid of before running it.

- Revert changes that were committed but weren't pushed

If you didn't yet push your changes, they currently live only on your local computer. So you're free to change things any way you like. Git has the command `git reset`, a form of which we saw above, to reset your current branch to something else. If you just want to get back to the state you were in before the commit, run

```
git reset HEAD^
```

which sets your current branch to the state before the most recent commit (that's what the ^ after `HEAD` means; you can put multiple `^`'s to go back more than one commit), and keeps your changes in the uncommitted state.

If you really want to get rid of the latest commit, `git reset` has a way to do that. If you run

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

the latest commit will be erased as if it never happened. Any code you had in the commit will be deleted, so only do this if you know the whole commit was a mistake. (You can put anything in place of `HEAD^` to reset the branch there; I sometimes use it to reset my local branch to whatever that branch is on the remote server because I want to get rid of anything I did locally on that branch.)

- Reverting changes committed and pushed to my own branch
In software development flows based on feature branches, it is common for a developer to have their "own" branch to work on the feature they're implementing. If no one else is working on a branch, you can treat the remote version of that branch pretty much the same as your local branch. So just do the git reset as above, followed by

```
git push --force
```

(called a force push) to make the remote match your local reseted version of the branch.

It is possible for a repository administrator to configure a remote repository to reject force pushes. If this is the case for you, follow the advice under the next heading.

- Reverting changes committed and pushed to a shared branch
The issue with doing a reset and force push on a branch on which multiple people are working is that some of those other people may have pulled the commits that shouldn't have happened and have already built some of their work on top of that. Reacting to a force push in such a situation can get hairy.

Luckily, git also has a way to revert changes without having to change the history. To get rid of the changes in the latest commit, run

```
git revert HEAD
```

This will create a new commit that is the exact opposite of the `HEAD` commit, thus undoing all the changes made there.

As you might expect, you can give any commit identifier to git revert in place of `HEAD`, and it does what you would expect, namely creates a commit that undoes only the commit you named. This is something you cannot do with a git reset, so it's a useful tool to remember even if your workflow wouldn't normally include using git revert.

- Revert changes that were committed and merged
Say you're working with the feature branch model, so any feature you're working on needs to be merged from your branch to the main development branch. It might happen that you or

someone accidentally merges that branch before it was ready to be merged, so you want to undo the merge.

Again, git revert can be used to undo a merge commit. But there's a catch: A merge commit has more than one parent (usually two). Which changes should git revert? The command to use to revert a merge commit is

```
git revert -m 1 <commit>
```

where the -m 1 option says to select parent number 1 as the parent to revert to.

Note that the parent number is not a commit identifier. Rather, a merge commit has a line

```
Merge: 8e2ce2d 86ac2e7
```

when you look at it with git log or git show. The parent number is the 1-based index of the desired parent on this line, that is, the first identifier is number 1, the second is number 2, and so on. Number 1 is the branch onto which the merge was made, so in a feature branch workflow, undoing a merge means giving option -m 1 to git revert.

- I committed and pushed something that shouldn't exist in the repository
Sometimes you keep stuff in your working tree that should not be committed. Do set up an appropriate `.gitignore` file for your project to avoid accidentally committing such stuff, but it may happen that something new appears that's not caught by your `.gitignore` and you accidentally end up committing and pushing it. This might be credentials that should be a secret, it might be massive binary files that just make using the repository uncomfortable, or it might be something else.

The way to go here is to follow the advise when you've committed and pushed to your own branch, that is, git reset followed by git push --force. But if you did this on a shared branch, remember to communicate to your team that this happened, so that they can adjust to a force push on a shared branch.

Finally, if you accidentally pushed private keys, credentials, passwords, or the like into a public repository, treat them as compromised. Immediately revoke any credentials, change any passwords, etc.

- Retrieving reverted code back
If the changes you made were just premature and you do want to make them at a later point, here are the ways to recover them based on how you got rid of them:

- `git checkout --`: If you did this, your changes are gone. Git never saw them so it cannot help you recover them.

- `git reset HEAD, git reset`: The changes remained in your working tree, just in a different state. If you remembered to save them somewhere, fetch them from there.
- `git reset --hard`: The changes are gone, but git saw them and might remember them. Look into git reflog (which is too big a topic to get into here, plus I couldn't give good advice on it anyway)
- `git revert`: The way to go is another git revert, this time giving the identifier of the commit that was created for the original revert. This gives you fun commit messages "Revert of "Revert of ...."", but it's the cleanest way to go. And if you're undoing a premature merge of a feature branch, remember to do this before merging the correct version of the feature branch. Otherwise, git will get confused as to which changes from the feature branch already exist on the main development branch.

- Adding Remote Shortcuts to Git

- Clone another user's repository:
  - `git clone https://github.com/miyagawa/cpanminus.git`
- Clone one of your own repositories:
  - `git clone git@github.com:hoelzro/linotify.git`
- To enable the following shortcuts:
  - `git clone github:miyagawa/cpanminus`
  - `git clone hoelzro:linotify`
- Add a URL section to your `.gitconfig`, with an `insteadOf` attribute that describes the prefix you'd like to use. Here's how the previous two examples look in my .gitconfig:

```
[url "git@github.com:hoelzro/"]
    insteadOf = hoelzro:
[url "https://github.com/"]
    insteadOf = github:
```

- Using git rm v/s rm:

- If you just use rm, in order to commit that the file was removed, you will need to follow it up with `git add <fileRemoved>/git rm <fileRemoved>`, so you might as well do it that way right off the bat. git rm does this in one step.
- Also, depending on your shell, doing git rm after having deleted the file, you will not get tab-completion so you'll have to spell out the path yourself, whereas if you git rm while the file still exists, tab completion will work as normal.
- To remove the file from the Git index (staging it for deletion on the next commit) but keep your copy on the local file system, you can use `git rm --cached`

- Copy changes from one branch to another

-- Using merge

```
git checkout BranchB
git merge BranchA
git push origin BranchB
```

- After the merge command, you will have some conflicts, which you will have to edit manually and fix.

-- Using rebase:

```
git checkout BranchB
git rebase BranchA
```

This takes BranchB and rebases it onto BranchA.

- Undo pushed commits

-- You can revert individual commits with:

```
git revert <commit_hash>
```

This will create a new commit which reverts the changes of the commit you specified. Note that it only reverts that specific commit, and not commits after that.

-- If you want to revert a range of commits, you can do it like this:

```
git revert <oldest_commit_hash>..<latest_commit_hash>
```

- List all the files in a commit

```
git show a303aa90779efdd2f6b9d90693e2cbbbe4613c1d
```

Although it lists the files, it also includes unwanted diff information for each.

```
git show --name-only <SHA1>
git log --name-only <SHA1>
```

- Checkout a commit

Same command as checking out a branch

```
git checkout <SHA1>
```

- Adding Git aliases

- Add the following to the `.gitconfig` file in your `$HOME` directory:

```
[alias]
  co = checkout
  ci = commit
  st = status
  br = branch
  hist = log --pretty=format:\"%h %ad | %s%d [%an]\" --graph --date=short
  type = cat-file -t
  dump = cat-file -p
```

- Also, if your shell supports aliases, or shortcuts, you can add aliases on this level, too. I use:

```
alias gs='git status '
alias ga='git add '
alias gb='git branch '
```

```
alias gc='git commit'
alias gd='git diff'
alias go='git checkout '
```

- Working with GitHub

```
- Install Git on your computer (comes default on macOS)
- Create a repository on your GitHub account
- git init
- git remote add origin <LINK_OF_GITHUB_REPOSITORY>
- git checkout -b master
- git add .
- git commit -m “initial commit”
- git push origin master
```

- What's the difference between HEAD^ and HEAD~ in Git?

Rules of thumb

Use ~ most of the time — to go back a number of generations, usually what you want
Use ^ on merge commits — because they have two or more (immediate) parents
Mnemonics:

Tilde ~ is almost linear in appearance and wants to go backward in a straight line
Caret ^ suggests an interesting segment of a tree or a fork in the road

Tilde

The “Specifying Revisions” section of the git rev-parse documentation defines ~ as `<rev>~<n>, e.g. master~3`
A suffix ~<n> to a revision parameter means the commit object that is the nth generation ancestor of the named commit object, following only the first parents. [For example,] <rev>~3 is equivalent to `<rev>^^^` which is equivalent to `<rev>^1^1^1` …

You can get to parents of any commit, not just HEAD. You can also move back through generations: for example, master~2 means the grandparent of the tip of the master branch, favoring the first parent on merge commits.

Caret

Git history is nonlinear: a directed acyclic graph (DAG) or tree. For a commit with only one parent, `rev~` and `rev^` mean the same thing. The caret selector becomes useful with merge commits because each one is the child of two or more parents — and strains language borrowed from biology.

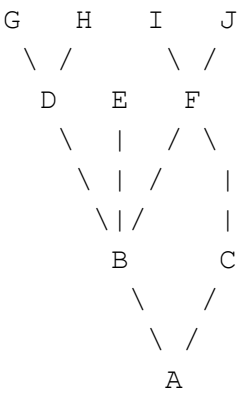
`HEAD^` means the first immediate parent of the tip of the current branch. `HEAD^` is short for `HEAD^1`, and you can also address `HEAD^2` and so on as appropriate. The same section of the git rev-parse documentation defines it as

`<rev>^`, e.g. `HEAD^, v1.5.1^0`
A suffix `^` to a revision parameter means the first parent of that commit object. ^<n> means the nth parent ([e.g.] `<rev>^` is equivalent to `<rev>^1`). As a special rule, `<rev>^0` means the commit itself and is used when `<rev>` is the object name of a tag object that refers to a commit object.

Examples
These specifiers or selectors can be chained arbitrarily, e.g., `topic~3^2` in English is the second parent of the merge commit that is the great-grandparent (three generations back) of the current tip of the branch topic.

The aforementioned section of the git rev-parse documentation traces many paths through a notional git history. Time flows generally downward. Commits `D`, `F`, `B`, and `A` are merge commits.

Here is an illustration, by Jon Loeliger. Both commit nodes `B` and `C` are parents of commit node `A`. Parent commits are ordered left-to-right.




```
A =       = A^0
B = A^    = A^1     = A~1
C = A^2
D = A^^   = A^1^1   = A~2
E = B^2   = A^^2
F = B^3   = A^^3
G = A^^^  = A^1^1^1 = A~3
H = D^2   = B^^2    = A^^^2  = A~2^2
I = F^    = B^3^    = A^^3^
J = F^2   = B^3^2   = A^^3^2
```

- How do I revert a Git repository to a previous commit?
This depends a lot on what you mean by "revert".

Temporarily switch to a different commit
If you want to temporarily go back to it, fool around, then come back to where you are, all you have to do is check out the desired commit:

```
# This will detach your HEAD, that is, leave you with no branch checked out:
git checkout 0d1d7fc32
```

Or if you want to make commits while you're there, go ahead and make a new branch while you're at it:

```
git checkout -b old-state 0d1d7fc32
```

To go back to where you were, just check out the branch you were on again. (If you've made changes, as always when switching branches, you'll have to deal with them as appropriate. You could reset to throw them away; you could stash, checkout, stash pop to take them with you; you could commit them to a branch there if you want a branch there.)

Hard delete unpublished commits
If, on the other hand, you want to really get rid of everything you've done since then, there are two possibilities. One, if you haven't published any of these commits, simply reset:

```
# This will destroy any local modifications.
# Don't do it if you have uncommitted work you want to keep.
git reset --hard 0d1d7fc32

# Alternatively, if there's work to keep:
git stash
git reset --hard 0d1d7fc32
git stash pop
# This saves the modifications, then reapplies that patch after resetting.
# You could get merge conflicts, if you've modified things which were
# changed since the commit you reset to.
```

If you mess up, you've already thrown away your local changes, but you can at least get back to where you were before by resetting again.

Undo published commits with new commits
On the other hand, if you've published the work, you probably don't want to reset the branch, since that's effectively rewriting history. In that case, you could indeed revert the commits. With Git, revert has a very specific meaning: create a commit with the reverse patch to cancel it out. This way you don't rewrite any history.

```
# This will create three separate revert commits:
git revert a867b4af 25eee4ca 0766c053

# It also takes ranges. This will revert the last two commits:
```

git revert HEAD~2..HEAD

#Similarly, you can revert a range of commits using commit hashes:
git revert a867b4af..0766c053

Reverting a merge commit
git revert -m 1 <merge_commit_sha>

To get just one, you could use `rebase -i` to squash them afterwards
Or, you could do it manually (be sure to do this at top level of the repo)
get your index and work tree into the desired state, without changing HEAD:
git checkout 0d1d7fc32 .

Then commit. Be sure and write a good message describing what you just did
git commit

- Using git revert
git revert makes a new commit
git revert simply creates a new commit that is the opposite of an existing commit.

It leaves the files in the same state as if the commit that has been reverted never existed. For example, consider the following simple example:

```
$ cd /tmp/example
$ git init
Initialized empty Git repository in /tmp/example/.git/
$ echo "Initial text" > README.md
$ git add README.md
$ git commit -m "initial commit"
[master (root-commit) 3f7522e] initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
$ echo "bad update" > README.md
$ git commit -am "bad update"
[master a1b9870] bad update
 1 file changed, 1 insertion(+), 1 deletion(-)
In this example the commit history has two commits and the last one
is a mistake. Using git revert:

$ git revert HEAD
[master 1db4eeb] Revert "bad update"
 1 file changed, 1 insertion(+), 1 deletion(-)
```

There will be 3 commits in the log:

$ git log --oneline
1db4eeb Revert "bad update"
a1b9870 bad update
3f7522e initial commit
So there is a consistent history of what has happened, yet the files are as if the bad update never occured:

cat README.md
Initial text
It doesn't matter where in the history the commit to be reverted is (in the above example, the last commit is reverted - any commit can be reverted).

- Throw away all your uncommitted changes using git reset --hard HEAD
First, it's always worth noting that git reset --hard is a potentially dangerous command, since it throws away all your uncommitted changes. For safety, you should always check that the output of git status is clean (that is, empty) before using it.

Initially you say the following:

So I know that Git tracks changes I make to my application, and it holds on to them until I commit the changes, but here's where I'm hung up:

That's incorrect. Git only records the state of the files when you stage them (with git add) or when you create a commit. Once you've created a commit which has your project files in a particular state, they're very safe, but until then Git's not really "tracking changes" to your files. (for example, even if you do git add to stage a new version of the file, that overwrites the previously staged version of that file in the staging area.)

In your question you then go on to ask the following:

When I want to revert to a previous commit I use: git reset --hard HEAD And git returns: HEAD is now at 820f417 micro

How do I then revert the files on my hard drive back to that previous commit?

If you do git reset --hard <SOME-COMMIT> then Git will:

Make your current branch (typically master) back to point at <SOME-COMMIT>.
Then make the files in your working tree and the index ("staging area") the same as the versions committed in <SOME-COMMIT>.
HEAD points to your current branch (or current commit), so all that git reset --hard HEAD will do is to throw away any uncommitted changes you have.

So, suppose the good commit that you want to go back to is f414f31. (You can find that via git log or any history browser.) You then have a few different options depending on exactly what you want to do:

Change your current branch to point to the older commit instead. You could do that with git reset --hard f414f31. However, this is rewriting the history of your branch, so you should avoid it if you've shared this branch with anyone. Also, the commits you did after f414f31 will no longer be in the history of your master branch.
Create a new commit that represents exactly the same state of the project as f414f31, but just adds that on to the history, so you don't lose any history. You can do that using the steps suggested in this answer - something like:

git reset --hard f414f31
git reset --soft HEAD@{1}
git commit -m "Reverting to the state of the project at f414f31"

- git pull till a particular commit
- git pull is nothing but git fetch followed by git merge. So what you can do is:
$ git fetch remote example_branch
$ git merge <commit_hash>

- First, fetch the latest commits from the remote repo. This will not affect your local branch.
git fetch origin

Then checkout the remote tracking branch and do a git log to see the commits

$ git checkout origin/master
$ git log

Grab the commit hash of the commit you want to merge up to (or just the first ~5 chars of it) and merge that commit into master
$ git checkout master
$ git merge <commit hash>

- To rebase onto the third commit from the current HEAD
$ git rebase HEAD~3

- How much of a git sha is *generally* considered necessary to uniquely identify a change in a given codebase?
This question is actually answered in Chapter 7 of the Pro Git book:

Generally, eight to ten characters are more than enough to be unique within a project. One of the largest Git projects, the Linux kernel, is beginning to need 12 characters out of the possible 40 to stay unique.

The first 7 digits is the Git default for a short SHA, so that's fine for most projects. The Kernel team have increased theirs several times, as mentioned, because the have several hundred thousand commits. So for your ~30k commits, 8 or 10 digits should be perfectly fine.

Note: you can ask git rev-parse --short for the shortest and yet unique SHA1.
See "git get short hash from regular hash"

$ git rev-parse --short=4 921103db8259eb9de72f42db8b939895f5651489
92110

- How do I remove Git tracking from a project?
All the data Git uses for information is stored in `.git/`, so removing it should work just fine. Of course, make sure that your working copy is in the exact state that you want it, because everything else will be lost. `.git` folder is hidden so make sure you turn on the `Show hidden files, folders and disks option`.

From there, you can run `git init` to create a fresh repository.

- Remove checked-in/local commit
- Use: `git log` to find the commit you want to remove. Copy commit hash.
- Use: `git rebase -i your_commit_hash_code_comes_here`

Just drop the commit you don't need and save the file.
- Push changes to server: `git push -f`
Interactive git rebase can let you also fix the broken commit - there is no need to remove it.

- Use '`git reset --hard HEAD`' to revert to a previous commit
`git reset --hard` is a potentially dangerous command, since it throws away all your uncommitted changes. For safety, you should always check that the output of git status is clean (that is, empty) before using it.

Git only records the state of the files when you stage them (with git add) or when you create a commit. Once you've created a commit which has your project files in a particular state, they're very safe, but until then Git's not really "tracking changes" to your files. (for example, even if you do git add to stage a new version of the file, that overwrites the previously staged version of that file in the staging area.)

When I want to revert to a previous commit I use: `git reset --hard HEAD` And git returns: `HEAD is now at 820f417 micro`

How do I then revert the files on my hard drive back to that previous commit?

If you do `git reset --hard <SOME-COMMIT>` then Git will:

Make your current branch (typically master) back to point at `<SOME-COMMIT>`.
Then make the files in your working tree and the index ("staging area") the same as the versions committed in `<SOME-COMMIT>`.
`HEAD` points to your current branch (or current commit), so all that `git reset --hard HEAD` will do is to throw away any uncommitted changes you have.

So, suppose the good commit that you want to go back to is f414f31. (You can find that via git log or any history browser.) You then have a few different options depending on exactly what you want to do:

Change your current branch to point to the older commit instead. You could do that with git reset --hard f414f31. However, this is rewriting the history of your branch, so you should avoid it if you've shared this branch with anyone. Also, the commits you did after f414f31 will no longer be in the history of your master branch.
Create a new commit that represents exactly the same state of the project as f414f31, but just adds that on to the history, so you don't lose any history. You can do that using the steps suggested in this answer - something like:

```
git reset --hard f414f31
git reset --soft HEAD@{1}
git commit -m "Reverting to the state of the project at f414f31"
```

- What is HEAD in Git?
You can think of the HEAD as the "current branch". When you switch branches with git checkout, the HEAD revision changes to point to the tip of the new branch.

You can see what HEAD points to by doing:
```
cat .git/HEAD
```

In my case, the output is:
```
$ cat .git/HEAD
ref: refs/heads/master
```

It is possible for HEAD to refer to a specific revision that is not associated with a branch name. This situation is called a detached HEAD.
In more practical terms, it can be thought of as a symbolic reference to the checked-out commit.