Flight Rules for Git

A guide for astronauts (now, programmers using Git) about what to do when things go wrong.

Kate Hudson & the contributors

Table of Contents

1 Abstract

‘Flight Rules are the hard-earned body of knowledge recorded in manuals that list, step-by-step, what to do if X occurs, and why. Essentially, they are extremely detailed, scenario-specific standard operating procedures. NASA has been capturing our missteps, disasters and solutions since the early 1960s, when Mercury-era ground teams first started gathering *lessons learned* into a compendium that now lists thousands of problematic situations, from engine failure to busted hatch handles to computer glitches, and their solutions.’

— **Chris Hadfield** - *An Astronaut’s Guide to Life*

2 Foreword

For clarity's sake all examples in this document use a customized bash prompt in order to indicate the current branch and whether or not there are staged changes. The branch is enclosed in parentheses, and a * next to the branch name indicates staged changes.

All commands should work for at least git version 2.13.0. See the git website to update your local git version.

3 Flight Rules

3.1 Repositories

3.1.1 I want to start a local repository

To initialize an existing directory as a Git repository:

```
(my-folder) $ git init
```

3.1.2 I want to clone a remote repository

To clone (copy) a remote repository, copy the url for the repository, and run:

```
$ git clone [url]
```

This will save it to a folder named the same as the remote repository's. Make sure you have connection to the remote server you are cloning from (for most purposes this means making sure you are connected to the internet).

To clone it into a folder with a different name than the default repository name:

```
$ git clone [url] name-of-new-folder
```

3.1.3 I set the wrong remote repository

There are a few possible problems here:

If you cloned the wrong repository, simply delete the directory created after running `git clone` and clone the correct repository.

If you set the wrong repository as the origin of an existing local repository, change the url of your origin by running:

```
$ git remote set-url origin [url of the actual repo]
```

For more, see this StackOverflow topic.

3.2 Editing Commits

3.2.1 What did I just commit?

Let's say that you just blindly committed changes with `git commit -a` and you're not sure what the actual content of the commit you just made was. You can show the latest commit on your current HEAD with:

```
(master)$ git show
```

Or

```
$ git log -n1 -p
```

If you want to see a file at a specific commit, you can also do this (where `<commitid>` is the commit you're interested in):

```
$ git show <commitid>:filename
```

3.2.2 I wrote the wrong thing in a commit message

If you wrote the wrong thing and the commit has not yet been pushed, you can do the following to change the commit message without changing the changes in the commit:

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

This will open your default text editor, where you can edit the message. On the other hand, you can do this all in one command:

```
$ git commit --amend --only -m 'xxxxxxx'
```

If you have already pushed the message, you can amend the commit and force push, but this is not recommended.

3.2.3 I committed with the wrong name and email configured

If it's a single commit, amend it

```
$ git commit --amend --no-edit --author "New Authorname <
    authoremail@mydomain.com>"
```

An alternative is to correctly configure your author settings in `git config --global author.(name|email)` and then use

```
$ git commit --amend --reset-author --no-edit
```

If you need to change all of history, see the man page for `git filter-branch`.

3.2.4 I want to remove a file from the previous commit

In order to remove changes for a file from the previous commit, do the following:

```
$ git checkout HEAD^ myfile
$ git add myfile
$ git commit --amend --no-edit
```

In case the file was newly added to the commit and you want to remove it (from Git alone), do:

```
$ git rm --cached myfile
$ git commit --amend --no-edit
```

This is particularly useful when you have an open patch and you have committed an unnecessary file, and need to force push to update the patch on a remote. The `--no-edit` option is used to keep the existing commit message.

3.2.5 I want to delete or remove my last commit

If you need to delete pushed commits, you can use the following. However, it will irreversibly change your history, and mess up the history of anyone else who had already pulled from the repository. In short, if you're not sure, you should never do this, ever.

```
$ git reset HEAD^ --hard
$ git push --force-with-lease [remote] [branch]
```

If you haven't pushed, to reset Git to the state it was in before you made your last commit (while keeping your staged changes):

```
(my-branch*)$ git reset --soft HEAD@{1}
```

This only works if you haven't pushed. If you have pushed, the only truly safe thing to do is `git revert SHAofBadCommit`. That will create a new commit that undoes all the previous commit's changes. Or, if the branch you pushed to is rebase-safe (ie. other devs aren't expected to pull from it), you can just use `git push --force-with-lease`. For more, see the above section.

3.2.6 Delete/remove arbitrary commit

The same warning applies as above. Never do this if possible.

```
$ git rebase --onto SHA1_OF_BAD_COMMIT^ SHA1_OF_BAD_COMMIT
$ git push --force-with-lease [remote] [branch]
```

Or do an interactive rebase and remove the line(s) corresponding to commit(s) you want to see removed.

3.2.7 I tried to push my amended commit to a remote, but I got an error message

```
To https://github.com/yourusername/repo.git
! [rejected]        mybranch -> mybranch (non-fast-forward)
error: failed to push some refs to 'https://github.com/tanay1337/webmaker.
    org.git'
hint: Updates were rejected because the tip of your current branch is
    behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Note that, as with rebasing (see below), amending **replaces the old commit with a new one**, so you must force push (`--force-with-lease`) your changes if you have already pushed the pre-amended commit to your remote. Be careful when you do this – *always* make sure you specify a branch!

```
(my-branch)$ git push origin mybranch --force-with-lease
```

In general, **avoid force pushing**. It is best to create and push a new commit rather than force-pushing the amended commit as it will cause conflicts in the source history for any other developer who has interacted with the branch in question or any child branches. `--force-with-lease` will still fail, if someone else was also working on the same branch as you, and your push would overwrite those changes.

If you are *absolutely* sure that nobody is working on the same branch or you want to update the tip of the branch *unconditionally*, you can use `--force` (`-f`), but this should be avoided in general.

3.2.8 I accidentally did a hard reset, and I want my changes back

If you accidentally do `git reset --hard`, you can normally still get your commit back, as git keeps a log of everything for a few days.

Note: This is only valid if your work is backed up, i.e., either committed or stashed. `git reset --hard` *will remove* uncommitted modifications, so use it with caution. (A safer option is `git reset --keep` .)

```
(master)$ git reflog
```

You'll see a list of your past commits, and a commit for the reset. Choose the SHA of the commit you want to return to, and reset again:

```
(master)$ git reset --hard SHA1234
```

And you should be good to go.

3.2.9 I accidentally committed and pushed a merge

If you accidentally merged a feature branch to the main development branch before it was ready to be merged, you can still undo the merge. But there's a catch: A merge commit has more than one parent (usually two).

The command to use

```
(feature-branch)$ git revert -m 1 <commit>
```

where the -m 1 option says to select parent number 1 (the branch into which the merge was made) as the parent to revert to.

Note: the parent number is not a commit identifier. Rather, a merge commit has a line `Merge: 8e2ce2d 86ac2e7`. The parent number is the 1-based index of the desired parent on this line, the first identifier is number 1, the second is number 2, and so on.

3.2.10 I accidentally committed and pushed files containing sensitive data

If you accidentally pushed files containing sensitive, or private data (passwords, keys, etc.), you can amend the previous commit. Keep in mind that once you have pushed a commit, you should consider any data it contains to be compromised. These steps can remove the sensitive data from your public repo or your local copy, but you **cannot** remove the sensitive data from other people's pulled copies.

If you committed a password, **change it immediately**. If you committed a key, **re-generate it immediately**. Amending the pushed commit is not enough, since anyone could have pulled the original commit containing your sensitive data in the meantime.

If you edit the file and remove the sensitive data, then run

```
(feature-branch)$ git add edited_file
(feature-branch)$ git commit --amend --no-edit
(feature-branch)$ git push --force-with-lease origin [branch]
```

If you want to remove an entire file (but keep it locally), then run

```
(feature-branch)$ git rm --cached sensitive_file
echo sensitive_file >> .gitignore
(feature-branch)$ git add .gitignore
(feature-branch)$ git commit --amend --no-edit
(feature-branch)$ git push --force-with-lease origin [branch]
```

Alternatively store your sensitive data in local environment variables.

If you want to completely remove an entire file (and not keep it locally), then run

```
(feature-branch)$ git rm sensitive_file
(feature-branch)$ git commit --amend --no-edit
(feature-branch)$ git push --force-with-lease origin [branch]
```

If you have made other commits in the meantime (i.e. the sensitive data is in a commit before the previous commit), you will have to rebase.

3.2.11 I want to remove a large file from ever existing in repo history

If the file you want to delete is secret or sensitive, instead see how to remove sensitive files.

Even if you delete a large or unwanted file in a recent commit, it still exists in git history, in your repo's `.git` folder, and will make `git clone` download unneeded files.

The actions in this part of the guide will require a force push, and rewrite large sections of repo history, so if you are working with remote collaborators, check first that any local work of theirs is pushed.

There are two options for rewriting history, the built-in `git-filter-branch` or `bfg-repo-cleaner`. `bfg` is significantly cleaner and more performant, but it is a third-party download and requires java. We will describe both alternatives. The final step is to force push your changes, which requires special consideration on top of a regular force push, given that a great deal of repo history will have been permanently changed.

3.2.11.1 Recommended Technique: Use third-party bfg

Using bfg-repo-cleaner requires java. Download the bfg jar from the link here. Our examples will use `bfg.jar`, but your download may have a version number, e.g. `bfg-1.13.0.jar`.

To delete a specific file.

```
(master)$ git rm path/to/filetoremove
(master)$ git commit -m "Commit removing filetoremove"
(master)$ java -jar ~/Downloads/bfg.jar --delete-files filetoremove
```

Note that in bfg you must use the plain file name even if it is in a subdirectory.

You can also delete a file by pattern, e.g.:

```
(master)$ git rm *.jpg
(master)$ git commit -m "Commit removing *.jpg"
(master)$ java -jar ~/Downloads/bfg.jar --delete-files *.jpg
```

With bfg, the files that exist on your latest commit will not be affected. For example, if you had several large .tga files in your repo, and then in an earlier commit, you deleted a subset of them, this call does not touch files present in the latest commit

Note, if you renamed a file as part of a commit, e.g. if it started as `LargeFileFirstName.mp4` and a commit changed it to `LargeFileSecondName.mp4`, running `java -jar ~/Downloads/bfg.jar --delete-files LargeFileSecondName.mp4` will not remove it from git history. Either run the `--delete-files` command with both filenames, or with a matching pattern.

3.2.11.2 Built-in Technique: Use git-filter-branch

`git-filter-branch` is more cumbersome and has less features, but you may use it if you cannot install or run `bfg`.

In the below, replace `filepattern` may be a specific name or pattern, e.g. `*.jpg`. This will remove files matching the pattern from all history and branches.

```
(master)$ git filter-branch --force --index-filter 'git rm --cached --
   ignore-unmatch filepattern' --prune-empty --tag-name-filter cat -- --
   all
```

Behind-the-scenes explanation:

`--tag-name-filter cat` is a cumbersome, but simplest, way to apply the original tags to the new commits, using the command cat.

`--prune-empty` removes any now-empty commits.

3.2.11.3 Final Step: Pushing your changed repo history

Once you have removed your desired files, test carefully that you haven't broken anything in your repo - if you have, it is easiest to re-clone your repo to start over. To finish, optionally use git garbage collection to minimize your local .git folder size, and then force push.

```
(master)$ git reflog expire --expire=now --all && git gc --prune=now --
   aggressive
(master)$ git push origin --force --tags
```

Since you just rewrote the entire git repo history, the `git push` operation may be too large, and return the error `"The remote end hung up unexpectedly"`. If this happens, you can try increasing the git post buffer:

```
(master)$ git config http.postBuffer 524288000
(master)$ git push --force
```

If this does not work, you will need to manually push the repo history in chunks of commits. In the command below, try increasing `<number>` until the push operation succeeds.

```
(master)$ git push -u origin HEAD~<number>:refs/head/master --force
```

Once the push operation succeeds the first time, decrease `<number>` gradually until a conventional `git push` succeeeds.

3.2.12 I need to change the content of a commit which is not my last

Consider you created some (e.g. three) commits and later realize you missed doing something that belongs contextually into the first of those commits. This bothers you, because if you'd create a new commit containing those changes, you'd have a clean code base, but your commits weren't atomic (i.e. changes that belonged to each other weren't in the same commit). In such a situation you may want to change the commit where these changes belong to, include them and have the following commits unaltered. In such a case, `git rebase` might save you.

Consider a situation where you want to change the third last commit you made.

```
(your-branch)$ git rebase -i HEAD~4
```

gets you into interactive rebase mode, which allows you to edit any of your last three commits. A text editor pops up, showing you something like

```
pick 9e1d264 The third last commit
pick 4b6e19a The second to last commit
pick f4037ec The last commit
```

which you change into

```
edit 9e1d264 The third last commit
pick 4b6e19a The second to last commit
pick f4037ec The last commit
```

This tells rebase that you want to edit your third last commit and keep the other two unaltered. Then you'll save (and close) the editor. Git will then start to rebase. It stops on the commit you want to alter, giving you the chance to edit that commit. Now you can apply the changes which you missed applying when you initially commited that commit. You do so by editing and staging them. Afterwards you'll run

```
(your-branch)$ git commit --amend
```

which tells Git to recreate the commit, but to leave the commit message unedited. Having done that, the hard part is solved.

```
(your-branch)$ git rebase --continue
```

will do the rest of the work for you.

3.3 Staging

3.3.1 I need to add staged changes to the previous commit

```
(my-branch*)$ git commit --amend
```

If you already know you don't want to change the commit message, you can tell git to reuse the commit message:

```
(my-branch*)$ git commit --amend -C HEAD
```

3.3.2 I want to stage part of a new file, but not the whole file

Normally, if you want to stage part of a file, you run this:

```
$ git add --patch filename.x
```

`-p` will work for short. This will open interactive mode. You would be able to use the `s` option to split the commit - however, if the file is new, you will not have this option. To add a new file, do this:

```
$ git add -N filename.x
```

Then, you will need to use the `e` option to manually choose which lines to add. Running `git diff --cached` or `git diff --staged` will show you which lines you have staged compared to which are still saved locally.

3.3.3 I want to add changes in one file to two different commits

`git add` will add the entire file to a commit. `git add -p` will allow to interactively select which changes you want to add.

3.3.4 I staged too many edits, and I want to break them out into a separate commit

`git reset -p` will open a patch mode reset dialog. This is similar to `git add -p`, except that selecting 'yes' will unstage the change, removing it from the upcoming commit.

3.3.5 I want to stage my unstaged edits, and unstage my staged edits

This is tricky. The best I figure is that you should stash your unstaged edits. Then, reset. After that, pop your stashed edits back, and add them.

```
$ git stash -k
$ git reset --hard
$ git stash pop
$ git add -A
```

3.4 Unstaged Edits

3.4.1 I want to move my unstaged edits to a new branch

```
$ git checkout -b my-branch
```

3.4.2 I want to move my unstaged edits to a different, existing branch

```
$ git stash
$ git checkout my-branch
$ git stash pop
```

3.4.3 I want to discard my local uncommitted changes (staged and unstaged)

If you want to discard all your local staged and unstaged changes, you can do this:

```
(my-branch)$ git reset --hard
# or
(master)$ git checkout -f
```

This will unstage all files you might have staged with `git add`:

```
$ git reset
```

This will revert all local uncommitted changes (should be executed in repo root):

```
$ git checkout .
```

You can also revert uncommitted changes to a particular file or directory:

```
$ git checkout [some_dir|file.txt]
```

Yet another way to revert all uncommitted changes (longer to type, but works from any subdirectory):

```
$ git reset --hard HEAD
```

This will remove all local untracked files, so only files tracked by Git remain:

```
$ git clean -fd
```

`-x` will also remove all ignored files.

3.4.4 I want to discard specific unstaged changes

When you want to get rid of some, but not all changes in your working copy.

Checkout undesired changes, keep good changes.

```
$ git checkout -p
# Answer y to all of the snippets you want to drop
```

Another strategy involves using `stash`. Stash all the good changes, reset working copy, and reapply good changes.

```
$ git stash -p
# Select all of the snippets you want to save
$ git reset --hard
$ git stash pop
```

Alternatively, stash your undesired changes, and then drop stash.

```
$ git stash -p
# Select all of the snippets you don't want to save
$ git stash drop
```

3.4.5 I want to discard specific unstaged files

When you want to get rid of one specific file in your working copy.

```
$ git checkout myFile
```

Alternatively, to discard multiple files in your working copy, list them all.

```
$ git checkout myFirstFile mySecondFile
```

3.4.6 I want to discard only my unstaged local changes

When you want to get rid of all of your unstaged local uncommitted changes

```
$ git checkout .
```

3.4.7 I want to discard all of my untracked files

When you want to get rid of all of your untracked files

```
$ git clean -f
```

3.4.8 I want to unstage a specific staged file

Sometimes we have one or more files that accidentally ended up being staged, and these files have not been committed before. To unstage them:

```
$ git reset -- <filename>
```

This results in unstaging the file and make it look like it's untracked.

3.5 Branches

3.5.1 I want to list all branches

List local branches

```
$ git branch
```

List remote branches

```
$ git branch -r
```

List all branches (both local and remote)

```
$ git branch -a
```

3.5.2 Create a branch from a commit

```
$ git checkout -b <branch> <SHA1_OF_COMMIT>
```

3.5.3 I pulled from/into the wrong branch

This is another chance to use `git reflog` to see where your HEAD pointed before the bad pull.

```
(master)$ git reflog
ab7555f HEAD@{0}: pull origin wrong-branch: Fast-forward
c5bc55a HEAD@{1}: checkout: checkout message goes here
```

Simply reset your branch back to the desired commit:

```
$ git reset --hard c5bc55a
```

Done.

3.5.4 I want to discard local commits so my branch is the same as one on the server

Confirm that you haven't pushed your changes to the server.

`git status` should show how many commits you are ahead of origin:

```
(my-branch)$ git status
# On branch my-branch
# Your branch is ahead of 'origin/my-branch' by 2 commits.
#   (use "git push" to publish your local commits)
#
```

One way of resetting to match origin (to have the same as what is on the remote) is to do this:

```
(master)$ git reset --hard origin/my-branch
```

3.5.5 I committed to master instead of a new branch

Create the new branch while remaining on master:

```
(master)$ git branch my-branch
```

Reset the branch master to the previous commit:

```
(master)$ git reset --hard HEAD^
```

`HEAD^` is short for `HEAD^1`. This stands for the first parent of `HEAD`, similarly `HEAD^2` stands for the second parent of the commit (merges can have 2 parents).

Note that `HEAD^2` is **not** the same as `HEAD~2` (see this link for more information).

Alternatively, if you don't want to use `HEAD^`, find out what the commit hash you want to set your master branch to (`git log` should do the trick). Then reset to that hash. `git push` will make sure that this change is reflected on your remote.

For example, if the hash of the commit that your master branch is supposed to be at is `a13b85e`:

```
(master)$ git reset --hard a13b85e
HEAD is now at a13b85e
```

Checkout the new branch to continue working:

```
(master)$ git checkout my-branch
```

3.5.6 I want to keep the whole file from another ref-ish

Say you have a working spike (see note), with hundreds of changes. Everything is working. Now, you commit into another branch to save that work:

```
(solution)$ git add -A && git commit -m "Adding all changes from this
   spike into one big commit."
```

When you want to put it into a branch (maybe feature, maybe `develop`), you're interested in keeping whole files. You want to split your big commit into smaller ones.

Say you have:

- branch `solution`, with the solution to your spike. One ahead of `develop`.
- branch `develop`, where you want to add your changes.

You can solve it bringing the contents to your branch:

```
(develop)$ git checkout solution -- file1.txt
```

This will get the contents of that file in branch `solution` to your branch `develop`:

```
# On branch develop
# Your branch is up-to-date with 'origin/develop'.
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#         modified:   file1.txt
```

Then, commit as usual.

Note: Spike solutions are made to analyze or solve the problem. These solutions are used for estimation and discarded once everyone gets clear visualization of the problem. ~ Wikipedia.

3.5.7 I made several commits on a single branch that should be on different branches

Say you are on your master branch. Running `git log`, you see you have made two commits:

```
(master)$ git log

commit e3851e817c451cc36f2e6f3049db528415e3c114
Author: Alex Lee <alexlee@example.com>
Date:   Tue Jul 22 15:39:27 2014 -0400
```

```
    Bug #21 - Added CSRF protection

commit 5ea51731d150f7ddc4a365437931cd8be3bf3131
Author: Alex Lee <alexlee@example.com>
Date:   Tue Jul 22 15:39:12 2014 -0400

    Bug #14 - Fixed spacing on title

commit a13b85e984171c6e2a1729bb061994525f626d14
Author: Aki Rose <akirose@example.com>
Date:   Tue Jul 21 01:12:48 2014 -0400

    First commit
```

Let's take note of our commit hashes for each bug (`e3851e8` for #21, `5ea5173` for #14).

First, let's reset our master branch to the correct commit (`a13b85e`):

```
(master)$ git reset --hard a13b85e
HEAD is now at a13b85e
```

Now, we can create a fresh branch for our bug #21:

```
(master)$ git checkout -b 21
(21)$
```

Now, let's *cherry-pick* the commit for bug #21 on top of our branch. That means we will be applying that commit, and only that commit, directly on top of whatever our head is at.

```
(21)$ git cherry-pick e3851e8
```

At this point, there is a possibility there might be conflicts. See the **There were conflicts** section in the interactive rebasing section above for how to resolve conflicts.

Now let's create a new branch for bug #14, also based on master

```
(21)$ git checkout master
(master)$ git checkout -b 14
(14)$
```

And finally, let's cherry-pick the commit for bug #14:

```
(14)$ git cherry-pick 5ea5173
```

3.5.8 I want to delete local branches that were deleted upstream

Once you merge a pull request on GitHub, it gives you the option to delete the merged branch in your fork. If you aren't planning to keep working on the branch, it's cleaner to delete the local copies of the branch so you don't end up cluttering up your working checkout with a lot of stale branches.

```
$ git fetch -p upstream
```

where, `upstream` is the remote you want to fetch from.

3.5.9 I accidentally deleted my branch

If you're regularly pushing to remote, you should be safe most of the time. But still sometimes you may end up deleting your branches. Let's say we create a branch and create a new file:

```
(master)$ git checkout -b my-branch
(my-branch)$ git branch
(my-branch)$ touch foo.txt
(my-branch)$ ls
README.md foo.txt
```

Let's add it and commit.

```
(my-branch)$ git add .
(my-branch)$ git commit -m 'foo.txt added'
(my-branch)$ foo.txt added
 1 files changed, 1 insertions(+)
 create mode 100644 foo.txt
(my-branch)$ git log

commit 4e3cd85a670ced7cc17a2b5d8d3d809ac88d5012
Author: siemiatj <siemiatj@example.com>
Date:   Wed Jul 30 00:34:10 2014 +0200

    foo.txt added

commit 69204cdf0acbab201619d95ad8295928e7f411d5
Author: Kate Hudson <katehudson@example.com>
Date:   Tue Jul 29 13:14:46 2014 -0400

    Fixes #6: Force pushing after amending commits
```

Now we're switching back to master and 'accidentally' removing our branch.

```
(my-branch)$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
(master)$ git branch -D my-branch
Deleted branch my-branch (was 4e3cd85).
(master)$ echo oh noes, deleted my branch!
oh noes, deleted my branch!
```

At this point you should get familiar with 'reflog', an upgraded logger. It stores the history of all the action in the repo.

```
(master)$ git reflog
69204cd HEAD@{0}: checkout: moving from my-branch to master
4e3cd85 HEAD@{1}: commit: foo.txt added
69204cd HEAD@{2}: checkout: moving from master to my-branch
```

As you can see we have commit hash from our deleted branch. Let's see if we can restore our deleted branch.

```
(master)$ git checkout -b my-branch-help
Switched to a new branch 'my-branch-help'
(my-branch-help)$ git reset --hard 4e3cd85
HEAD is now at 4e3cd85 foo.txt added
(my-branch-help)$ ls
README.md foo.txt
```

Voila! We got our removed file back. `git reflog` is also useful when rebasing goes terribly wrong.

3.5.10 I want to delete a branch

To delete a remote branch:

```
(master)$ git push origin --delete my-branch
```

You can also do:

```
(master)$ git push origin :my-branch
```

To delete a local branch:

```
(master)$ git branch -d my-branch
```

To delete a local branch that *has not* been merged to the current branch or an upstream:

```
(master)$ git branch -D my-branch
```

3.5.11 I want to delete multiple branches

Say you want to delete all branches that start with `fix/`:

```
(master)$ git branch | grep 'fix/' | xargs git branch -d
```

3.5.12 I want to rename a branch

To rename the current (local) branch:

```
(master)$ git branch -m new-name
```

To rename a different (local) branch:

```
(master)$ git branch -m old-name new-name
```

3.5.13 I want to checkout to a remote branch that someone else is working on

First, fetch all branches from remote:

```
(master)$ git fetch --all
```

Say you want to checkout to `daves` from the remote.

```
(master)$ git checkout --track origin/daves
Branch daves set up to track remote branch daves from origin.
Switched to a new branch 'daves'
```

(`--track` is shorthand for `git checkout -b [branch] [remotename]/[branch]`)

This will give you a local copy of the branch `daves`, and any update that has been pushed will also show up remotely.

3.5.14 I want to create a new remote branch from current local one

```
$ git push <remote> HEAD
```

If you would also like to set that remote branch as upstream for the current one, use the following instead:

```
$ git push -u <remote> HEAD
```

With the `upstream` mode and the `simple` (default in Git 2.0) mode of the `push.default` config, the following command will push the current branch with regards to the remote branch that has been registered previously with `-u`:

```
$ git push
```

The behavior of the other modes of `git push` is described in the doc of `push.default`.

3.5.15 I want to set a remote branch as the upstream for a local branch

You can set a remote branch as the upstream for the current local branch using:

```
$ git branch --set-upstream-to [remotename]/[branch]
# or, using the shorthand:
$ git branch -u [remotename]/[branch]
```

To set the upstream remote branch for another local branch:

```
$ git branch -u [remotename]/[branch] [local-branch]
```

3.5.16 I want to set my HEAD to track the default remote branch

By checking your remote branches, you can see which remote branch your HEAD is tracking. In some cases, this is not the desired branch.

```
$ git branch -r
  origin/HEAD -> origin/gh-pages
  origin/master
```

To change `origin/HEAD` to track `origin/master`, you can run this command:

```
$ git remote set-head origin --auto
origin/HEAD set to master
```

3.5.17 I made changes on the wrong branch

You've made uncommitted changes and realise you're on the wrong branch. Stash changes and apply them to the branch you want:

```
(wrong_branch)$ git stash
(wrong_branch)$ git checkout <correct_branch>
(correct_branch)$ git stash apply
```

3.6 Rebasing and Merging

3.6.1 I want to undo rebase/merge

You may have merged or rebased your current branch with a wrong branch, or you can't figure it out or finish the rebase/merge process. Git saves the original HEAD pointer in a variable called ORIG_HEAD before doing dangerous operations, so it is simple to recover your branch at the state before the rebase/merge.

```
(my-branch)$ git reset --hard ORIG_HEAD
```

3.6.2 I rebased, but I don't want to force push

Unfortunately, you have to force push, if you want those changes to be reflected on the remote branch. This is because you have changed the history. The remote branch won't accept changes unless you force push. This is one of the main reasons many people use a merge workflow, instead of a rebasing workflow - large teams can get into trouble with developers force pushing. Use this with caution. A safer way to use rebase is not to reflect your changes on the remote branch at all, and instead to do the following:

```
(master)$ git checkout my-branch
(my-branch)$ git rebase -i master
(my-branch)$ git checkout master
(master)$ git merge --ff-only my-branch
```

For more, see this SO thread.

3.6.3 I need to combine commits

Let's suppose you are working in a branch that is/will become a pull-request against `master`. In the simplest case when all you want to do is to combine *all* commits into a single one and you don't care about commit timestamps, you can reset and recommit. Make sure the master branch is up to date and all your changes committed, then:

```
(my-branch)$ git reset --soft master
(my-branch)$ git commit -am "New awesome feature"
```

If you want more control, and also to preserve timestamps, you need to do something called an interactive rebase:

```
(my-branch)$ git rebase -i master
```

If you aren't working against another branch you'll have to rebase relative to your `HEAD`. If you want to squash the last 2 commits, for example, you'll have to rebase against `HEAD~2`. For the last 3, `HEAD~3`, etc.

```
(master)$ git rebase -i HEAD~2
```

After you run the interactive rebase command, you will see something like this in your text editor:

```
pick a9c8a1d Some refactoring
pick 01b2fd8 New awesome feature
pick b729ad5 fixup
pick e3851e8 another fix

# Rebase 8074d12..b729ad5 onto 8074d12
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
```

```
#
# Note that empty commits are commented out
```

All the lines beginning with a `#` are comments, they won't affect your rebase.

Then you replace `pick` commands with any in the list above, and you can also remove commits by removing corresponding lines.

For example, if you want to **leave the oldest (first) commit alone and combine all the following commits with the second oldest**, you should edit the letter next to each commit except the first and the second to say `f`:

```
pick a9c8a1d Some refactoring
pick 01b2fd8 New awesome feature
f b729ad5 fixup
f e3851e8 another fix
```

If you want to combine these commits **and rename the commit**, you should additionally add an `r` next to the second commit or simply use `s` instead of `f`:

```
pick a9c8a1d Some refactoring
pick 01b2fd8 New awesome feature
s b729ad5 fixup
s e3851e8 another fix
```

You can then rename the commit in the next text prompt that pops up.

```
Newer, awesomer features

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# rebase in progress; onto 8074d12
# You are currently editing a commit while rebasing branch 'master' on '
    8074d12'.
#
# Changes to be committed:
#   modified:   README.md
#
```

If everything is successful, you should see something like this:

```
(master)$ Successfully rebased and updated refs/heads/master.
```

3.6.3.1 Safe merging strategy

`--no-commit` performs the merge but pretends the merge failed and does not autocommit, giving the user a chance to inspect and further tweak the merge result before committing. `no-ff` maintains evidence that a feature branch once existed, keeping project history consistent.

```
(master)$ git merge --no-ff --no-commit my-branch
```

3.6.3.2 I need to merge a branch into a single commit

```
(master)$ git merge --squash my-branch
```

3.6.3.3 I want to combine only unpushed commits

Sometimes you have several work in progress commits that you want to combine before you push them upstream. You don't want to accidentally combine any commits that have already been pushed upstream because someone else may have already made commits that reference them.

```
(master)$ git rebase -i @{u}
```

This will do an interactive rebase that lists only the commits that you haven't already pushed, so it will be safe to reorder/fix/squash anything in the list.

3.6.3.4 I need to abort the merge

Sometimes the merge can produce problems in certain files, in those cases we can use the option `abort` to abort the current conflict resolution process, and try to reconstruct the pre-merge state.

```
(my-branch)$ git merge --abort
```

This command is available since Git version >= 1.7.4

3.6.4 I need to update the parent commit of my branch

Say I have a master branch, a feature-1 branch branched from master, and a feature-2 branch branched off of feature-1. If I make a commit to feature-1, then the parent commit of feature-2 is no longer accurate (it should be the head of feature-1, since we branched off of it). We can fix this with `git rebase --onto`.

```
(feature-2)$ git rebase --onto feature-1 <the first commit in your feature
   -2 branch that you don't want to bring along> feature-2
```

This helps in sticky scenarios where you might have a feature built on another feature that hasn't been merged yet, and a bugfix on the feature-1 branch needs to be reflected in your feature-2 branch.

3.6.5 Check if all commits on a branch are merged

To check if all commits on a branch are merged into another branch, you should diff between the heads (or any commits) of those branches:

```
(master)$ git log --graph --left-right --cherry-pick --oneline HEAD...
   feature/120-on-scroll
```

This will tell you if any commits are in one but not the other, and will give you a list of any nonshared between the branches. Another option is to do this:

```
(master)$ git log master ^feature/120-on-scroll --no-merges
```

3.6.6 Possible issues with interactive rebases

3.6.6.1 The rebase editing screen says 'noop'

If you're seeing this:

```
noop
```

That means you are trying to rebase against a branch that is at an identical commit, or is *ahead* of your current branch. You can try:

- making sure your master branch is where it should be
- rebase against `HEAD~2` or earlier instead

3.6.6.2 There were conflicts

If you are unable to successfully complete the rebase, you may have to resolve conflicts.

First run `git status` to see which files have conflicts in them:

```
(my-branch)$ git status
On branch my-branch
```

```
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
     )

  both modified:   README.md
```

In this example, `README.md` has conflicts. Open that file and look for the following:

```
   <<<<<<< HEAD
   some code
   =========
   some code
   >>>>>>> new-commit
```

You will need to resolve the differences between the code that was added in your new commit (in the example, everything from the middle line to **`new`**`-commit`) and your `HEAD`.

If you want to keep one branch's version of the code, you can use `--ours` or `--theirs`:

```
(master*)$ git checkout --ours README.md
```

- When *merging*, use `--ours` to keep changes from the local branch, or `--theirs` to keep changes from the other branch.
- When *rebasing*, use `--theirs` to keep changes from the local branch, or `--ours` to keep changes from the other branch. For an explanation of this swap, see this note in the Git documentation.

If the merges are more complicated, you can use a visual diff editor:

```
(master*)$ git mergetool -t opendiff
```

After you have resolved all conflicts and tested your code, `git add` the files you have changed, and then continue the rebase with `git rebase --`**`continue`**

```
(my-branch)$ git add README.md
(my-branch)$ git rebase --continue
```

If after resolving all the conflicts you end up with an identical tree to what it was before the commit, you need to `git rebase --skip` instead.

If at any time you want to stop the entire rebase and go back to the original state of your branch, you can do so:

```
(my-branch)$ git rebase --abort
```

3.7 Stash

3.7.1 Stash all edits

To stash all the edits in your working directory

```
$ git stash
```

If you also want to stash untracked files, use `-u` option.

```
$ git stash -u
```

3.7.2 Stash specific files

To stash only one file from your working directory

```
$ git stash push working-directory-path/filename.ext
```

To stash multiple files from your working directory

```
$ git stash push working-directory-path/filename1.ext working-directory-
    path/filename2.ext
```

3.7.3 Stash with message

```
$ git stash save <message>
```

3.7.4 Apply a specific stash from list

First check your list of stashes with message using

```
$ git stash list
```

Then apply a specific stash from the list using

```
$ git stash apply "stash@{n}"
```

Here, 'n' indicates the position of the stash in the stack. The topmost stash will be position 0.

3.8 Finding

3.8.1 I want to find a string in any commit

To find a certain string which was introduced in any commit, you can use the following structure:

```
$ git log -S "string to find"
```

Commons parameters:

- `--source` means to show the ref name given on the command line by which each commit was reached.
- `--all` means to start from every branch.
- `--reverse` prints in reverse order, it means that will show the first commit that made the change.

3.8.2 I want to find by author/committer

To find all commits by author/committer you can use:

```
$ git log --author=<name or email>
$ git log --committer=<name or email>
```

Keep in mind that author and committer are not the same. The `--author` is the person who originally wrote the code; on the other hand, the `--committer`, is the person who committed the code on behalf of the original author.

3.8.3 I want to list commits containing specific files

To find all commits containing a specific file you can use:

```
$ git log -- <path to file>
```

You would usually specify an exact path, but you may also use wild cards in the path and file name:

```
$ git log -- **/*.js
```

While using wildcards, it's useful to inform `--name-status` to see the list of committed files:

```
$ git log --name-status -- **/*.js
```

3.8.4 I want to view the commit history for a specific function

To trace the evolution of a single function you can use:

```
$ git log -L :FunctionName:FilePath
```

Note that you can combine this with further `git log` options, like revision ranges and commit limits.

3.8.5 Find a tag where a commit is referenced

To find all tags containing a specific commit:

```
$ git tag --contains <commitid>
```

3.9 Submodules

3.9.1 Clone all submodules

```
$ git clone --recursive git://github.com/foo/bar.git
```

If already cloned:

```
$ git submodule update --init --recursive
```

3.9.2 Remove a submodule

Creating a submodule is pretty straight-forward, but deleting them less so. The commands you need are:

```
$ git submodule deinit submodulename
$ git rm submodulename
$ git rm --cached submodulename
$ rm -rf .git/modules/submodulename
```

3.10 Miscellaneous Objects

3.10.1 Restore a deleted file

First find the commit when the file last existed:

```
$ git rev-list -n 1 HEAD -- filename
```

Then checkout that file:

```
git checkout deletingcommitid^ -- filename
```

3.10.2 Delete tag

```
$ git tag -d <tag_name>
$ git push <remote> :refs/tags/<tag_name>
```

3.10.3 Recover a deleted tag

If you want to recover a tag that was already deleted, you can do so by following these steps: First, you need to find the unreachable tag:

```
$ git fsck --unreachable | grep tag
```

Make a note of the tag's hash. Then, restore the deleted tag with following, making use of `git update-ref`:

```
$ git update-ref refs/tags/<tag_name> <hash>
```

Your tag should now have been restored.

3.10.4 Deleted Patch

If someone has sent you a pull request on GitHub, but then deleted their original fork, you will be unable to clone their repository or to use `git am` as the .diff, .patch urls become unavailable. But you can checkout the PR itself using GitHub's special refs. To fetch the content of PR#1 into a new branch called pr_1:

```
$ git fetch origin refs/pull/1/head:pr_1
From github.com:foo/bar
```

```
 * [new ref]         refs/pull/1/head -> pr_1
```

3.10.5 Exporting a repository as a Zip file

```
$ git archive --format zip --output /full/path/to/zipfile.zip master
```

3.10.6 Push a branch and a tag that have the same name

If there is a tag on a remote repository that has the same name as a branch you will get the following error when trying to push that branch with a standard `$ git push <remote> <branch>` command.

```
$ git push origin <branch>
error: dst refspec same matches more than one.
error: failed to push some refs to '<git server>'
```

Fix this by specifying you want to push the head reference.

```
$ git push origin refs/heads/<branch-name>
```

If you want to push a tag to a remote repository that has the same name as a branch, you can use a similar command.

```
$ git push origin refs/tags/<tag-name>
```

3.11 Tracking Files

3.11.1 I want to change a file name's capitalization, without changing the contents of the file

```
(master)$ git mv --force myfile MyFile
```

3.11.2 I want to overwrite local files when doing a git pull

```
(master)$ git fetch --all
(master)$ git reset --hard origin/master
```

3.11.3 I want to remove a file from Git but keep the file

```
(master)$ git rm --cached log.txt
```

3.11.4 I want to revert a file to a specific revision

Assuming the hash of the commit you want is c5f567:

```
(master)$ git checkout c5f567 -- file1/to/restore file2/to/restore
```

If you want to revert to changes made just 1 commit before c5f567, pass the commit hash as c5f567~1:

```
(master)$ git checkout c5f567~1 -- file1/to/restore file2/to/restore
```

3.11.5 I want to list changes of a specific file between commits or branches

Assuming you want to compare last commit with file from commit c5f567:

```
$ git diff HEAD:path_to_file/file c5f567:path_to_file/file
```

Same goes for branches:

```
$ git diff master:path_to_file/file staging:path_to_file/file
```

3.11.6 I want Git to ignore changes to a specific file

This works great for config templates or other files that require locally adding credentials that shouldn't be committed.

```
$ git update-index --assume-unchanged file-to-ignore
```

Note that this does *not* remove the file from source control - it is only ignored locally. To undo this and tell Git to notice changes again, this clears the ignore flag:

```
$ git update-index --no-assume-unchanged file-to-stop-ignoring
```

3.12 Debugging with Git

The git-bisect command uses a binary search to find which commit in your Git history introduced a bug.

Suppose you're on the `master` branch, and you want to find the commit that broke some feature. You start bisect:

```
$ git bisect start
```

Then you should specify which commit is bad, and which one is known to be good. Assuming that your *current* version is bad, and `v1.1.1` is good:

```
$ git bisect bad
$ git bisect good v1.1.1
```

Now `git-bisect` selects a commit in the middle of the range that you specified, checks it out, and asks you whether it's good or bad. You should see something like:

```
$ Bisecting: 5 revision left to test after this (roughly 5 step)
$ [c44abbbee29cb93d8499283101fe7c8d9d97f0fe] Commit message
$ (c44abbb)$
```

You will now check if this commit is good or bad. If it's good:

```
$ (c44abbb)$ git bisect good
```

and `git-bisect` will select another commit from the range for you. This process (selecting `good` or `bad`) will repeat until there are no more revisions left to inspect, and the command will finally print a description of the **first** bad commit.

3.13 Configuration

3.13.1 I want to add aliases for some Git commands

On OS X and Linux, your git configuration file is stored in `~/.gitconfig`. I've added some example aliases I use as shortcuts (and some of my common typos) in the `[alias]` section as shown below:

```
[alias]
    a = add
    amend = commit --amend
    c = commit
```

```
    ca = commit --amend
    ci = commit -a
    co = checkout
    d = diff
    dc = diff --changed
    ds = diff --staged
    extend = commit --amend -C HEAD
    f = fetch
    loll = log --graph --decorate --pretty=oneline --abbrev-commit
    m = merge
    one = log --pretty=oneline
    outstanding = rebase -i @{u}
    reword = commit --amend --only
    s = status
    unpushed = log @{u}
    wc = whatchanged
    wip = rebase -i @{u}
    zap = fetch -p
    day = log --reverse --no-merges --branches=* --date=local --since=
       midnight --author=\"$(git config --get user.name)\"
    delete-merged-branches = "!f() { git checkout --quiet master && git
       branch --merged | grep --invert-match '\\*' | xargs -n 1 git branch
        --delete; git checkout --quiet @{-1}; }; f"
```

3.13.2 I want to add an empty directory to my repository

You can't! Git doesn't support this, but there's a hack. You can create a .gitignore file in the directory with the following contents:

```
# Ignore everything in this directory
*
# Except this file
!.gitignore
```

Another common convention is to make an empty file in the folder, titled .gitkeep.

```
$ mkdir mydir
$ touch mydir/.gitkeep
```

You can also name the file as just .keep , in which case the second line above would be `touch mydir/.keep`

3.13.3 I want to cache a username and password for a repository

You might have a repository that requires authentication. In which case you can cache a username and password so you don't have to enter it on every push and pull. Credential helper can do this for you.

```
$ git config --global credential.helper cache
# Set git to use the credential memory cache
```

```
$ git config --global credential.helper 'cache --timeout=3600'
# Set the cache to timeout after 1 hour (setting is in seconds)
```

To find a credential helper:

```
$ git help -a | grep credential
# Shows you possible credential helpers
```

For OS specific credential caching:

```
$ git config --global credential.helper osxkeychain
# For OSX
```

```
$ git config --global credential.helper manager
# Git for Windows 2.7.3+
```

```
$ git config --global credential.helper gnome-keyring
# Ubuntu and other GNOME-based distros
```

More credential helpers can likely be found for different distributions and operating systems.

3.13.4 I want to make Git ignore permissions and filemode changes

```
$ git config core.fileMode false
```

If you want to make this the default behaviour for logged-in users, then use:

```
$ git config --global core.fileMode false
```

3.13.5 I want to set a global user

To configure user information used across all local repositories, and to set a name that is identifiable for credit when review version history:

```
$ git config --global user.name "[firstname lastname]"
```

To set an email address that will be associated with each history marker:

```
git config --global user.email "[valid-email]"
```

3.13.6 I want to add command line coloring for Git

To set automatic command line coloring for Git for easy reviewing:

```
$ git config --global color.ui auto
```

3.14 I've no idea what I did wrong

So, you're screwed - you `reset` something, or you merged the wrong branch, or you force pushed and now you can't find your commits. You know, at some point, you were doing alright, and you want to go back to some state you were at.

This is what `git reflog` is for. `reflog` keeps track of any changes to the tip of a branch, even if that tip isn't referenced by a branch or a tag. Basically, every time HEAD changes, a new entry is added to the reflog. This only works for local repositories, sadly, and it only tracks movements (not changes to a file that weren't recorded anywhere, for instance).

```
(master)$ git reflog
0a2e358 HEAD@{0}: reset: moving to HEAD~2
0254ea7 HEAD@{1}: checkout: moving from 2.2 to master
c10f740 HEAD@{2}: checkout: moving from master to 2.2
```

The reflog above shows a checkout from master to the 2.2 branch and back. From there, there's a hard reset to an older commit. The latest activity is represented at the top labeled `HEAD@{0}`.

If it turns out that you accidentally moved back, the reflog will contain the commit master pointed to (0254ea7) before you accidentally dropped 2 commits.

```
$ git reset --hard 0254ea7
```

Using `git reset` it is then possible to change master back to the commit it was before. This provides a safety net in case history was accidentally changed.

(copied and edited from Source).

3.15 Git Shortcuts

3.15.1 Git Bash

Once you're comfortable with what the above commands are doing, you might want to create some shortcuts for Git Bash. This allows you to work a lot faster by doing complex tasks in really short commands.

```
alias sq=squash

function squash() {
    git rebase -i HEAD~$1
}
```

Copy those commands to your .bashrc or .bash_profile.

3.15.2 PowerShell on Windows

If you are using PowerShell on Windows, you can also set up aliases and functions. Add these commands to your profile, whose path is defined in the `$profile` variable. Learn more at the About Profiles page on the Microsoft documentation site.

```
Set-Alias sq Squash-Commits

function Squash-Commits {
  git rebase -i HEAD~$1
}
```

4 Other Resources

4.1 Books

- Learn Enough Git to Be Dangerous - A book by Michael Hartl covering Git from basics
- Pro Git - Scott Chacon and Ben Straub's excellent book about Git
- Git Internals - Scott Chacon's other excellent book about Git

4.2 Tutorials

- 19 Git Tips For Everyday Use - A list of useful Git one liners
- Atlassian's Git tutorial Get Git right with tutorials from beginner to advanced.
- Learn Git branching An interactive web based branching/merging/rebasing tutorial
- Getting solid at Git rebase vs. merge
- Git Commands and Best Practices Cheat Sheet - A Git cheat sheet in a blog post with more explanations
- Git from the inside out - A tutorial that dives into Git's internals
- git-workflow - Aaron Meurer's howto on using Git to contribute to open source repositories
- GitHub as a workflow - An interesting take on using GitHub as a workflow, particularly with empty PRs
- Githug - A game to learn more common Git workflows
- learnGitBranching - An interactive git visualization to challenge and educate!

4.3 Scripts and Tools

- firstaidgit.io A searchable selection of the most frequently asked Git questions
- git-extra-commands - a collection of useful extra Git scripts
- git-extras - GIT utilities – repo summary, repl, changelog population, author commit percentages and more

- git-fire - git-fire is a Git plugin that helps in the event of an emergency by adding all current files, committing, and pushing to a new branch (to prevent merge conflicts).
- git-tips - Small Git tips
- git-town - Generic, high-level Git workflow support! http://www.git-town.com

4.4 GUI Clients

- GitKraken - The downright luxurious Git client,for Windows, Mac & Linux
- git-cola - another Git client for Windows and OS X
- GitUp - A newish GUI that has some very opinionated ways of dealing with Git's complications
- gitx-dev - another graphical Git client for OS X
- Sourcetree - Simplicity meets power in a beautiful and free Git GUI. For Windows and Mac.
- Tower - graphical Git client for OS X (paid)
- tig - terminal text-mode interface for Git
- Magit - Interface to Git implemented as an Emacs package.
- GitExtensions - a shell extension, a Visual Studio 2010-2015 plugin and a standalone Git repository tool.
- Fork - a fast and friendly Git client for Mac (beta)
- gmaster - a Git client for Windows that has 3-way merge, analyze refactors, semantic diff and merge (beta)
- gitk - a Git client for linux to allow simple view of repo state.
- SublimeMerge - Blazing fast, extensible client that provides 3-way merges, powerful search and syntax highlighting, in active development.