Git Cheat Sheet

I wrote this because I have a serious case of monkey brain and forget how `git rebase` works every 30 days. Anyways.

Porcelain

Configs

Git can be configured from several locations. These include:

- system: `/etc/gitconfig`, a file that configures Git for all users on the system
- global: `~/.gitconfig` , a file that configures Git for all projects of one user
- local: `.git/config` , a file that configures Git for a specific project
- worktree: `.git/config/worktree` , a file that configures Git for a portion of a project.

Most of the time you work with the global configurations to set things like username and password, and the local configurations to set preferences specific to a project.

`git config --add --global user.name "poopysnoop"` This is like the main global command you need to set things like username, email, and password (maybe), but you can also change the flag to `--local` to keep it local. Obviously. The `--add` shockingly indicates that we are adding a configuration. `--global` indications that we want this stored globally, i.e. applying to all repos everywhere. `user` is the section and `name` is the key within the section, and finally "poopysnoop" is the value we want stored under that key. You can use `--unset` to remove the value for any key.

Commits

`git commit --amend -m ""` will change the commit message in case you forget to capitalize the first letter or swear. don't tell the fia. but keep in mind that this will change the SHA of your commit. whatever that means.

I think if you do `git commit --amend --no-edit`, then you can add changes to the commit without changing the commit message. So like if you make a commit and realize you are missing one line of code, you can add the line to the file and then run that command. I think. Don't quote me on this.

Squashing

Squashing takes a series of commits and "squashes" them into one commit. Whats weird that is squashing is actually done using `git rebase`.

Here are the steps to squash the last `n` commits:

1. Start an interactive rebase using `git rebase -i HEAD~n`, where `n` is the number of commits you want to squash.

2. Git will open your default editor with a list of commits. Change the word `pick` to `squash` for all but the first commit.
3. Save and close the editor.

The `-i` flag stands for "interactive," and it allows us to edit the commit history before Git applies the changes. `HEAD~n` is how we reference the last `n` commits. `HEAD` points to the current commit (as long as we're in a clean state) and `~n` means "n commits before HEAD". So why `rebase`? Remember that `rebase` replays changes, so when we rebase onto a specific commit, or `HEAD~n` as you will see shortly, this replays all the changes from the current branch onto that commit. The `-i` flag is what allows us to squash those changes and apply them onto a single commit.

Note: squashing is a destructive operation. You remove commit history of the project, and can no longer go back to those commits even though the changes stay there. You can always go through reflog and whatnot to get it back but it's a pain - don't do that to yourself.

Logs

`git --no-pager log`no-pager will output the log directly into sdout. so no scrolling! lit

`git log -n 10` will show the log of just the last 10 commits. use it with no-pager.

`git log --oneline` this makes it so much more readable! seems like most of the time you want to use this

`git log --graph` will make a graph. add the flag `--all` to see all branches in the log, not just the one you are on. add the flag `--parents` to see the parents commit for each commit in the graph. add the `--decorate` flag to show the tags and branch names. idk what tags are tho.

Reflog

`git reflog` is similar to `git log` but stands for reference log, and logs the changes to a reference that have happened over time. A reference is pretty much just a pointer (i'm pretty sure). For example, `HEAD` is a reference to the branch that you are currently on. So if you are on `main`, `HEAD` is just a pointer to `main`. Overall it seems like reflog just tracks where your `HEAD` is at throughout the lifespan of your repo.

Reflog shows the logs as steps back in time from the current branch or `HEAD`. For example:

- `HEAD@{0}` is a where `HEAD` is at the present moment.
- `HEAD@{1}` is where `HEAD` was 1 move ago
- `HEAD@{2}` is where `HEAD` was 2 moves ago

And so on.

Reflog saves every step you make along the journey of writing your ultimately useless code. For example, suppose you have 3 commits, but you mistakenly use `git reset --hard HEAD~1` and "delete" your third commit. Running `git log` will show you two entries representing your two remaining commits. Running `git reflog` will actually show four entries: your three commits and the command moving back to the second commit. This means that even after running `git reset --hard`, if you decide you actually do need

your content from your deleted third commit, you can use `git reflog` and some of the plumbing to get that work back. Idk how to do that though, probably something with blobs and flobs or whatever.

A easier way to do it (blobless) is using `git merge COMMITISH` This command takes in something called a commitish, which is anything that looks like a commit, including branches, tags (i still don't know what these are) and `HEAD@{1}`. So, if you want to recover that third commit you mistakenly deleted in the paragraph above, you can simply run `git merge HEAD@{1}` , or `git merge tag` where tag is a tag (i think this is the like unique 7 start of the hash? maybe?) (do i finally know what a tag is?!) (no not really go look at the tags section)

Branches

A branch is just a named pointer to a specific commit. Creating a branch creates a new pointer to that commit, and the commit that the branch points to is called the tip of the branch. Note that this is generally the most recent commit made to the branch.

Creating Branches

`git branch new_branch` will create a new branch, but will not move the `HEAD` to that branch, so most people do the following command instead

`git switch -c new_branch` will create a new branch (`-c` specifies this) and then switch to that branch as well.

When you create a new branch, the current commit that you are on is the branch base. Meaning that right at branch creation, main and new_branch are pointers to the same commit until you make new commits to either branch.

You can use `git checkout branch_name` or `git checkout -b branch_name` to create and move between branches as well, but this ages you. The youth use `switch`.

`git switch -c new_branch COMMITHASH` allows you to branch off of a specific commit using the hash.

Renaming Branches

`git branch -m oldname newname` renames a branch from `oldname` to `newname`. m for rename???? k

Deleting Branches

`git branch -d branch_name` deletes the branch.

Merging

Merge commits are the only commits with two parents. Suppose you have two branches, `main` and `feature`. Merging `feature` into `main` from `main` will find the merge base commit (the best common ancestor of the two branches), first add all the changes on `main` since this ancestor on a new commit, then add all the changes from `feature` into this new commit, and finally commit the new commit with all the changes onto `main`. This means that the commits that were on `main` since the merge base will come before the commits that were on `feature` on the final merge commit.

Fast forward merges are the simplest merges. This occurs when the branch we are merging into `main` has ALL of the commits that `main` has. Generally this means that there are no new commits to main that our new branch does not have. In this case, Git just moves the pointer of the "base" branch to the tip of the new branch. In our running example, this means moving the pointer to the tip of `main` to the tip of `feature`.

A diagram for you visual learners (me) out there:

```
      C     feature
     /
A - B      main
```

will become

```
           feature
A - B - C  main
```

Merge Conflicts

Conflicts must be resolved manually. When you attempt to `merge` and run into a conflict, you will enter a conflict. In the command line output you'll see something like "You have unmerged paths" (allegedly).

One way to deal with the conflict is to manually resolve it. Open the marked files with the conflict. The top section, between the `<<<<<<< HEAD` and `=======` lines, is your branch's version of the file. The bottom section, between the `=======` and `>>>>>>> main` lines, is the version of the file that's on the `main` branch (or whatever other branch you are merging into). You can either delete one change and keep the other, (also deleting conflict markers: the `<` and `=`) or keeping both changes if you are dealing with stuff like comments or data. Then, add and commit your changes.

Another way to deal with the merge conflict is using the `git checkout` command. This can checkout the individual changes using the flags `--theirs` and `--ours`. The `--ours` flag will overwrite the file with the changes from the branch we are currently on and merging into, and `--theirs` flag will overwrite the file with changes from the branch you are merging into the current branch. The final command is:

```
git checkout --theirs path/to/file
```

or, of course, using the `--ours` flag.

Rebase (rebussy) ((based))

Rebasing allows you to alter the base commit that commits from a branch diverge from. Suppose you have branches `main` and `feature`. You make some commits on `feature`, and some commits on `main` and now you want to bring the changes from `main` onto `feature`. You could merge, but this creates a merge commit with some consequences for the git history that will not be delved into here. Yet. Assuming I update this later. Rebasing replays the commits from `feature` on top of `main`, allowing for a later fast-forward merge that does not create a merge commit. Rebasing does exactly what it sounds like - it changes the base commit that your branch originates from (aka changes the merge base of your branch). It does NOT

create any merges or alter the branch you rebase from. It simply allows you to bring in changes from one branch (`main`) into another (`feature`).

For example,

```
A — B — C    main
   \
    D — E    feature
```

will become

```
A — B — C         main
         \
          D — E   feature
```

To rebase and bring changes from `main` onto `feature`, we run `git rebase main` from `feature`. This does NOT affect the main branch, but brings all of the changes from main onto the current branch. Note that the parent commit for `D` is now `C` instead of `A`, meaning that the hashes changes for commits `D` and `E`.

You should NEVER rebase a public branch like `main` onto a personal branch. You can do what you want with your own branches, but the history of public branches must remain clean. You can however rebase your personal branches onto `main`.

Rebase Conflicts

Remember that when you rebase a branch into yours, for example if you run `git rebase main` from the `feature` branch, `rebase` checks out onto the source branch `main` so that it can replay all of `features` changes on top. This means that when you hit the conflict, `HEAD` is actually pointing to `main`. Consequently, the `--their` flag will actually refer to our own changes on `feature` , and `--ours` will refer to the changes on `main`. So you kind of have to think in reverse here.

When you resolve a rebase conflict, you are not on a branch, but are instead in the "detached HEAD" state. Like in merge conflicts, we can use `git checkout --theirs/--ours` to resolve the conflict, and then we add the files.

Once you have resolved the file, DO NOT COMMIT like you would in a merge. Instead, we continue the rebase using `git rebase –continue`.

Note: If you ever accidentally commit the resolution of a rebase conflict (committing something rather than running `git rebase --continue`, just run `git reset --soft HEAD~1` to take you back one commit (while still keeping all the changes) and continue the rebase.

RERERE (RERERERERERERERERERE)

When enabled `rerere` will remember how a conflict was resolved and automatically apply the resolution to the next time it sees the same conflict. This is useful when rebasing multiple branches into main. To enable it, use `git config --local rerere.enabled true`, with the corresponding flag.

Reset

Resetting can be used to undo the last commit(s), or any staged changes.

`git reset --soft COMMITHASH` The commit hash is the commit you want to reset back to. The --soft option only goes back one commit, but keeps all of your changes. Commit changes are uncommited and staged, and uncommited changes remain the way they were (either staged or unstaged).

`git reset --hard COMMITHASH` undoes all the changes that were staged, and goes back to the specified commit. Keep in mind though that `--hard` means losing your git tracking. If you were to simply delete a committed file, it would be trivially easy to recover because it is tracked in Git. However, if you used `git reset --hard` to undo committing that file, it would be deleted for good, because any changes are deleted as well.

Instead of using the commit hashes, you can also do `git reset --soft HEAD~1`, which would take you back one commit. `~2` takes you back two, and so on.

Revert

A revert is effectively an anti commit. It does not remove the commit (like reset), but instead creates a new commit that does the exact opposite of the commit being reverted. It undoes the change but keeps a full history of the change and its undoing.

To do a revert, you need the commit hash of the commit you want to revert, which you find via `git log`:

`git revert <commit-hash>` will complete the revert

When you are working alone on your own branch, using `git reset` is usually fine, but when working on a shared branch or when working with older changes, then it is safer to use `git revert`.

Diff

The `diff` command shows the differences between commits, the working tree, etc. It can be used to see changes between the current state of the code and the last commit.

`git diff` will show the changes between the working tree and the last commit.

`git diff HASH1 HASH2` will show the changes between two commits

Cherry Pick (yum)

Should you want to grab a commit from a branch, but do not want to merge or rebase because you don't want all the commits. The `cherry-pick` command allows you to do this. You can pick a commit off one branch and plop it on top of your own branch.

To cherry pick a command, the first thing you need to do is clear the working tree (ensure that you have no uncommitted changes). Then, run

`git cherry-pick <commit-hash>` with the commit hash of the commit you want to pick.

If you screw up your cherry-pick, you can do `git cherry-pick --abort`. Tbh I think the `--abort` flag can stop pretty much anything.

This confused me at first because we know that git stores a snapshot of the entire repo at each commit, not just the diffs. So I was like "wait, so if I have `A - B - C` on `main` and `A - B - C - D - E` on `feature`, and cherry-pick `E` onto `main`, do the changes from `D` also get plopped onto `main`? *No.* Git will fist calculate the diff between the cherry-picked commit and it's parent, and only apply the diff to the current head (in our case, `main`).

Git Bisect

Bisect is a tool to help you find a commit that introduced a bug (usually, but I guess you could find other stuff too). It uses binary search to do this. At a high level, you provide a commit where the known status was good, and a commit where the known status was bad, and then git bisect uses binary search to help you find the commit that changed the status of the branch from good to bad. Here are the steps to follow:

1. Start the bisect with `git bisect start`
2. Select a "good" commit with `git bisect good <commitish>` (a commit where you're sure the bug wasn't present)
3. Select a bad commit via `git bisect bad <commitish>` (a commit where you're sure the bug was present)
4. Git will checkout a commit between the good and bad commits for you to test to see if the bug is present
5. Execute `git bisect good` or `git bisect bad` to mark the current commit as either good or bad
6. Loop back to step 4 (until `git bisect` completes)
7. Exit the bisect mode with `git bisect reset`

There are lots of other things you can do like `next` and `skip` and stuff which I'm assuming will skip commits or pass over some or something but I'm not really going to get into that here.

But look! Git bisect can be automated! From `man git-bisect`:

```
Bisect run If you have a script that can tell if the current source code
is good or bad, you can bisect by issuing the command:

        $ git bisect run my_script arguments

   Note that the script (my_script in the above example) should exit with
code 0 if the
   current source code is good/old, and exit with a code between 1 and 127
(inclusive),
   except 125, if the current source code is bad/new.
```

So you can write a script to check the status, and automate away!

Worktrees

A worktree (or working directory) is the directory in your file-system where the files you are tracking with git are stored. In general this is the root of the git repo (where the `.git` directory is). This is referred to as the main worktree, but you can have more than one working tree.

Worktrees allow you to work on different changes without losing work or having to create branches or stash changes. The main worktree contains the `.git` directory with the entire state of the repo in it. Creating a new main working tree will require cloning the repo or running `git init`.

A linked worktree contains a `.git` file with a path to the main working tree. This is much lighter than the main worktree as it does not store as much data. To create a linked worktree, we run the following command from within the root of our current repo:

`git worktree add <path> [<branch>]` where `path` specifies where we want the worktree to reside. I think you usually want this worktree to live outside of your repo, so the path would be something like `../new-worktree`. When you create a worktree without giving a branch name manually, Git creates a new branch (named after the directory) and it points to your current `HEAD`.

`git worktree list` will list all the worktrees you have for a given repo.

Linked worktrees are just like any other git repo, but with one major difference: you cannot work on a branch that is currently checked out by any other working tree (main or linked). If you run `git branch` from within the linked worktree, any branches that are currently checked out by another worktree will be prefixed by a `+`.

When you make changes in a linked worktree, the change is automatically reflected in the main worktree. The linked worktree is not a new repo, but a different view of the same repo, almost like another branch in the repo but with is own space in the filesystem.

There are two ways to delete worktrees. The simplest way is `git worktree remove WORKTREE_NAME`. You can also delete the directory of the worktree manually and run `git worktree prune` which remove the references to any deleted directories.

Tags (finally)

A tag is a name linked to a commit that doesn't move between commits, unlike a branch. They can be created and deleted, but not modified.

`git tag` will list all current tags

`git tag -a "tag name" -m "tag message"` will create a tag on the current commit

`git push origin --tags` will push your tags up to your remote Github repo

Pretty much anywhere you can use a commit hash, you can use a tag name when working with the git CLI. It's a "commitish".

So I think a tag is just a name? Like you name your commits? But not like "Simon" or "Jane" or something, because that would be stupid. Seemingly random numbers seem like a much better option.

Semver

Semver is a naming convention for software, following the layout `v3.14.2`. In this example, 3 is the major increment, 14 is the minor increment, and 5 is the path increment. This allows us to understand how large the changes in each version are and how difficult it will be to update or install them.

Each part is a number that starts at 0 and increments upward forever. The rules are simple:

- MAJOR increments when we make "breaking" changes (this is typically a big release, for example, Python 2 -> Python 3)
- MINOR increments when we add new features in a backward-compatible manner
- PATCH increments when we make backward-compatible bug fixes

Okay I guess that actually makes sense.

Remote (github is NOT git guys)

In git, another repo is called a remote. Note that the remote does not necessarily have to be hosted on github or gitlab or whatever, it could literally just be another repo you have locally.

`git remote add <name> <uri>` is the command for adding a remote. Most of the time `origin` is passed in for the `name` argument, and the `uri` is either the relative path to the repo on your machine or the github link or whatever.

Fetch and Pull

Now that the remote is set, you must fetch the metadata of the remote. This is done using `git fetch`. This gathers all the objects, but does not create any branches or commits. To do this, you must merge a branch into your repo (or pull, i think this will do it too).

For example, `git merge origin/feature` will merge the feature branch into your repo from the origin.

`git log remote/branch` lets you look at the log for the branches in a remote. This is useful to decide which branches you may want to pull or merge into your repo.

`git push` sends local changes to any remote, usually github. `git push origin main` would push the commits from our `main` branch to the remote `origin`'s `main` branch. When you are pushing from a branch, for example if you want to push your changes from branch `feature`, make sure you then run `git push origin feature`, so that you are not just pushing to `main`.

`git pull` gives us the actual file changes, unlike `git fetch` which just gives us the metadata.

Pull Requests

Creating pull requests is done through the remote (usually github for me). So you open github, and create a pr using the UI. You pick the base branch, usually `main`, but this is just the branch you are planning to merge into, and then you compare it with the new feature you are proposing.

Force Push

`git push origin <branch> --force` is a useful but very dangerous command. It allows us to overwrite the remote branch with our local branch. It basically just says "make that branch the same as this branch". So you can see why it would piss off your co-workers if you do this. But fuck those guys anyways.

Git Stash

The `git stash` command records the current state of your working directory and the index (staging area) (idk what this is). It records the changes in a safe place and then reverts the working directory to match the `HEAD` commit (the last commit on the current branch).

`git stash list` will list all the stashes.

`git stash pop` will apply your most recent stash entry to the working directory and remove it from the stash list. Essentially this just undoes the `git stash` command.

`git stash -m "stash message"` will stash the changes with a message, like a commit.

`git stash apply` will apply the changes in your stash to your working directory without deleting the entry from the stash list .

`git stash drop` will delete the most recent stash from the stash list without applying any of the changes.

`git stash apply stash@{n}` will apply the `n`th most recent stash. If you run `git stash apply stash@{2}`, this applies the third most recent stash (0 indexing).

The stash is a LIFO stack, meaning that when you `pop` from the stash you will always get the most recently stashed item first. Both staged and unstaged changes can be stashed, which I'm told is very useful.

Stash is very useful when you have some changes but then need to pull from the main. In this case, you can

1. Stash your changes
2. Pull remote changes
3. Pop from the stash

allowing you to get all the changes from `main` and then continue working. If you get a conflict when applying your stash, it can be resolved in the same way as merge conflicts are.

Git ignore

The `.gitignore` file specifies files that should not be tracked by git. Suppose you add `dir_a` as a line to your `.gitignore`. Git will now ignore `dir_a/code.py`, `dir_b/dir_a/code.py`, and any other files in a directory called `dir_a`, or even files named `dir_a`. It will not ignore a directory or file titled `dir_ab`. You can have multiple `.gitignore` files across your repo. Each `.gitignore` only applies to the directory it is in and all subdirectories beneath it. Alternatively, imagine you have `content.md` in your `.gitignore`. This means that any file with this name in any subdirectory of your repo will be ignored.

There are some patterns that are useful in ignoring many files at once. These include:

- `*.txt` The `*` character matches to any number of characters except `/`. You can add other extensions like `.pdf`, or create longer paths to ignore stuff
- `/main.py` Patterns that start with `/` are anchored to the directory that contain the `.gitignore`. This ignores the `main.py` in the root directory, but not in an subdirectories.
- `!important.txt` The exclamation part is a negation, so this would ensure that `important.txt` is not ignored. This is useful if you want to ignore all `.txt` files except for `important.txt`

Remember that the order of patterns in a `.gitignore` file determines their effect, and patterns can override each other.

But what should be ignored? Not me. Please do not ignore me. Anyways, here are some rules of thumb:

1. things that can be generated (compiled code, etc) (i have no more examples) (maybe `.html` files from markdown?)
2. dependencies (`venv`, `packages`, the like)
3. things that are personal or specific to how you work (editor settings)
4. things that are dangerous (api keys, `.env` files, etc)

Note that the `.gitignore` file is shared by everyone in the repo. Therefore, if you want to ignore things that you are creating for yourself, (like notes, outfiles, test-files, whatever), these should go in the `/.git/info/exclude` file, so you don't all add your personal crap to the shared `.gitignore`.

Plumbing

All the data in a Git repository is stored directly in the hidden `.git` directory. Git is made up of objects that are stored in the `.git/objects` directory. Comits, branches, and tags are all objects.

`git cat-file -p <hash>` allows you to look at the contents of the commit. it seems to be utterly unreadable though

Terms to know:

- tree: how git stores a directory
- blob: how git stores a file

When we inspect a commit using cat-file, we can see the tree object, the author, the committer, and the commit message. We cannot see the contents of the modified files. These are stored in the blob objects.

Git does NOT store just the diffs, but an entire snapshot of files every commit. There are some optimizations that prevent the `.git` directory from becoming obscenely large, such as file compressions and de-duplications of files that are unchanged across commits. What does this mean exactly? Well, I am not glad you asked.

Suppose we have 2 files, `file_a` and `file_b`. We commit the contents of both of these files. This first commit will have a tree with a unique hash, and a unique hash for each blob. Now suppose we change only `file_b` and make a second commit. This second commit will have a new tree with its own hash, and the blob for `file_b` will also get a new hash. However, the hash for `file_a` will be reused. Git will always create a new tree for a commit, but it does not mean that it stores every file. For unchanged files, it will store a pointer/reference (idk if its just one of these or both) to that file, maintaining the history of the repo.

References

All of this is derivative of the boot.dev course on Git by The Primeagen. Some of it is word for word, some of it is paraphrased. This is for personal use only because otherwise it's just plagiarism and AI does enough of that already.