Introduction Tutorial to Git

QI Group Meeting

Yoann Piétri

10/02/2023

LIP6 - Sorbonne Université - CNRS

What is Git?

Git

Git is a free and open source **distributed version control system**.

> *It is also impossible to change any file, date, commit message, or any other data in a Git repository without changing the IDs of everything after it.*
>
> *The Pro Git book*

- Snapshots, Not Differences
- Nearly Every Operation Is Local *(Sorry non-locality)*
- Git Has Integrity *(Not like us)*
- Git Generally Only Adds Data
- The Three States (Working directory, Staging area, Repository)
- Git is cool

What is not Git?

Not Git

Git is not:

- A magic solution to everything;
- A backup system[1];
- A synchronisation system;
- Github, Gitlab or any other tool built on top of Git;

[1] In some specific cases, it may be used at such

Other version control tools

Git is not alone in the world of version controls and here are examples of some other tools:

- Subversion (svn)
- CVS
- BitKeeper
- Fossil
- RCS
- ...

Git remains the most use version control system today.

Good reads

Good reads on Git:

- The Pro Git book `https://git-scm.com/book/en/v2` (many things of the presentation are inspired from there)
- The git reference manual `https://git-scm.com/docs`

Git basics

Start of a project example

```
$ git init -b main
$ git config --local user.email "Yoann.Pietri@lip6.fr"
$ touch README.md
$ git add README.md
$ git commit -m "Initial commit"
```

1. Initialize the repository with branch `main`
2. Configure the local (for current git repository) email
3. Create a file name `README.md`
4. Add the file to the staging area (see later), i.e. what's going to be committed
5. Commit

What is a commit?

Commit

A commit is a **snapshot** (i.e. a picture) of your code at a given moment.

Unlike other version control systems, Git does not only store the difference between versions, each commit represents the full code, and git think you repository as a series of commits.

It makes git a miniature file system, with version control, integrity and powerful tools.

Snapshots, Not Differences

Integrity

Let's show the commit history of our simple earlier project:

```
$ git log
commit 13d588fd13f50a8554dfc6cb06d83b689c82ef81 (HEAD -> main)
Author: Yoann Piétri <Yoann.Pietri@lip6.fr>
Date:   Wed Dec 28 11:08:29 2022 +0100

Initial commit
```

What is `13d588fd13f50a8554dfc6cb06d83b689c82ef81` ?

Integrity

At commit, Git takes into his snapshot everything in the staging area, along with some metadata (commit name, committer information, date etc) and secure it in a secure manner and **generates a unique checksum**.

Any change on the commit will make this checksum change.

Integrity

Let's show the commit history of our simple earlier project:

```
$ git commit --amend -m "Almost intial commit"
$ git log
commit 95129c0bd7ed42a458773d9fa4d6406d268a6161 (HEAD -> main)
Author: Yoann Piétri <Yoann.Pietri@lip6.fr>
Date:   Wed Dec 28 11:08:29 2022 +0100

Almost intial commit
```

95129c0bd7ed42a458773d9fa4d6406d268a6161 $\neq$
13d588fd13f50a8554dfc6cb06d83b689c82ef81

Integrity

It is also impossible to change any file, date, commit message, or any other data in a Git repository without changing the IDs[2] of everything after it.

The Pro Git book

Git Has Integrity

[2] The checksums are also the IDs of the git objects

Not screwing things up

> *When you do actions in Git, nearly all of them only add data to the Git database. It is hard[3] to get the system to do anything that is not undoable or to make it erase data in any way.*
> *This makes using Git a joy because we know we can experiment without the danger of severely screwing things up[4].*

The Pro Git book

Git Generally Only Adds Data

[3] Hard doesn't mean impossible
[4] Still possible though

The (almost) 3 states

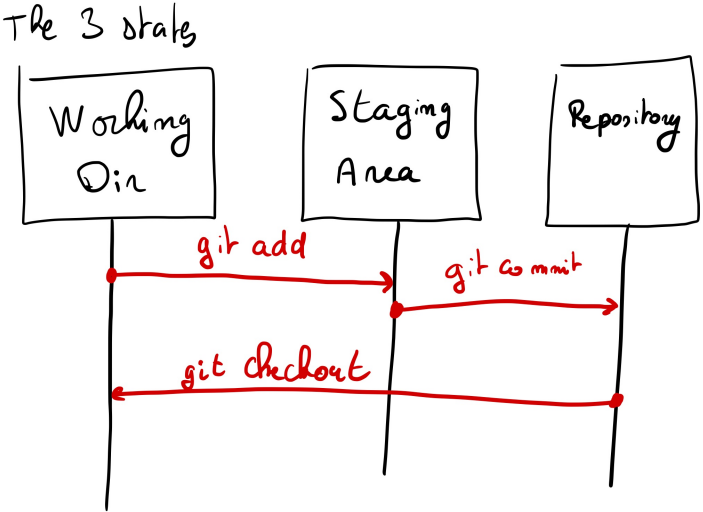
The 3 states

Git has 3 states and 3 associated regions

- Modified
- Staged
- Committed

- Working Directory
- Staging area
- Repository

The 3 states


The 3 states
Working Dir
Staging Area
Repository
git add
git commit
git checkout

The hidden state

What if, in the middle of implementing a feature, you need to work on another branch, or implement an hotfix ? Stashing is here for you.

> *Stashing takes the dirty state of your working directory and saves it on a stack of unfinished changes that you can reapply at any time (even on a different branch).*
>
> *The Pro Git book*

The hidden state

The hidden state

```
$ git status
    On branch main
    nothing to commit, working tree clean
$ touch unfinished_work
$ git status
    On branch main
    Untracked files:
        unfinished_work

$ ls
    README.md  unfinished_work
```

The hidden state

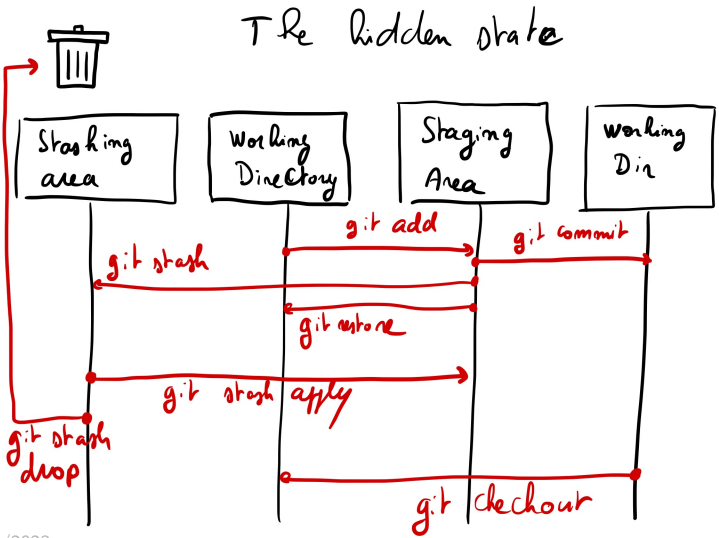
```
$ git add unfinished_work
$ git stash
    Saved working directory and index state WIP on main: b0f26d4 Initial commit
$ git status
    On branch main
    nothing to commit, working tree clean
$ ls
   README.md
$ git stash list
stash@{0}: WIP on main: b0f26d4 Initial commit
```

The hidden state

```
$ git stash apply
    On branch main
    Changes to be committed:
        new file:   unfinished_work
$ ls
    README.md  unfinished_work
```

```
$ git stash drop
    Dropped refs/stash@{0} (3fd9e24718f373740c72398833fb1be353586246)
```

The hidden state



The Three States

Pay attention now - here is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states [...]

The Pro Git book

The Three States

The forgotten modifications

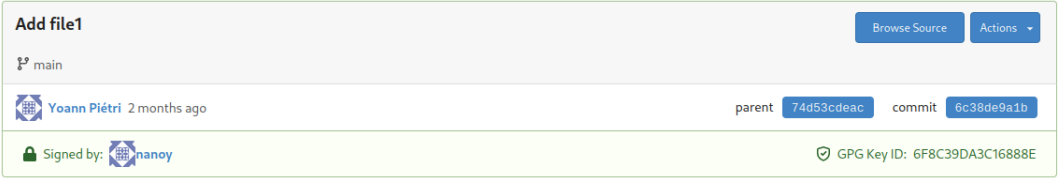
Branches

What is a commit?

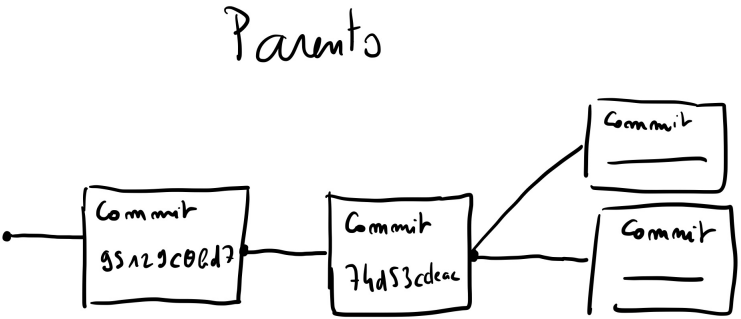
(déjà-vu ?)

Commit

A commit is a **snapshot** of your staged work along with some metadata for the commit (author, data, message) and **one ore more parent commit(s)**.



Branches


Parents
Commit
95129c0bd7
Commit
74d53cdeac
Commit
Commit

Branch

```
$ git checkout -b main2
    Switched to a new branch 'main2'
$ touch file2
$ git add file2
$ git commit -m "Add file2"
$ git checkout main
    Switched to branch 'main'
$ touch file1
$ git add file1
$ git commit -m "Add file1"
```

Branch

```
$ git log --graph --oneline --all
    * 6c38de9 (HEAD -> main) Add file1
    | * ed86875 (main2) Add file2
    |/
    * 95129c0 Almost intial commit
```

What is a branch?

Branch

A branch is a **lightwight movable pointer** to one of the project's commits.

When you commit on a branch, the pointer moves to the new commit.

When you create a commit from a branch A, you create an additional pointer B that points to the same commit as A, at the beginning.

What is HEAD?

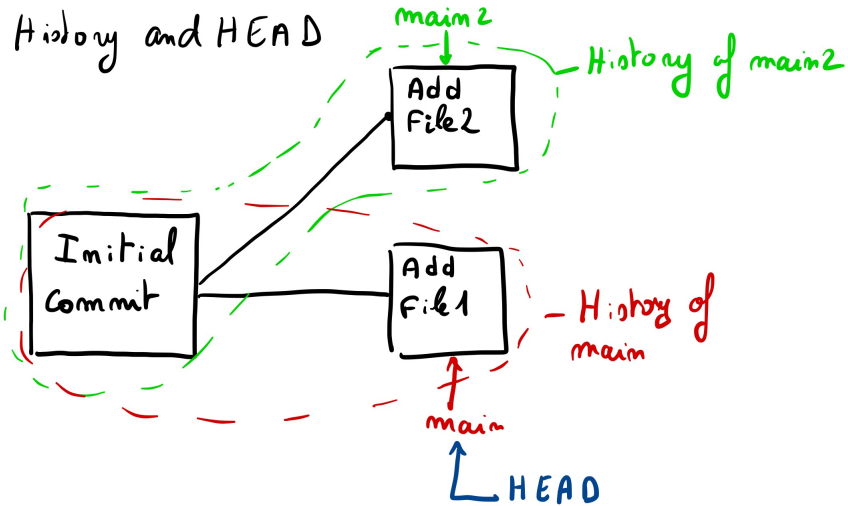
HEAD

Head is special pointer pointing to the current branch (could also point to a specific commit in certain case). It's also where the next commit will be added.

```
$ git log --graph --oneline --all
    * 6c38de9 (HEAD -> main) Add file1
    | * ed86875 (main2) Add file2
    |/
    * 95129c0 Almost intial commit
```

```
$ git checkout main2
$ git log --graph --oneline --all
    * 6c38de9 (main) Add file1
    | * ed86875 (HEAD -> main2) Add file2
    |/
    * 95129c0 Almost intial commit
```

HEAD


History and HEAD
main2
History of main2
Add File2
Initial Commit
Add File1
History of main
main
HEAD

Bringing back those branches together

Why branches?

> *Branching means you diverge from the main line of development and continue to do work without messing with that main line.*
>
> *The Pro Git book*

At some point, you might want to integrate your change in the main line of development.

Merge

Merging

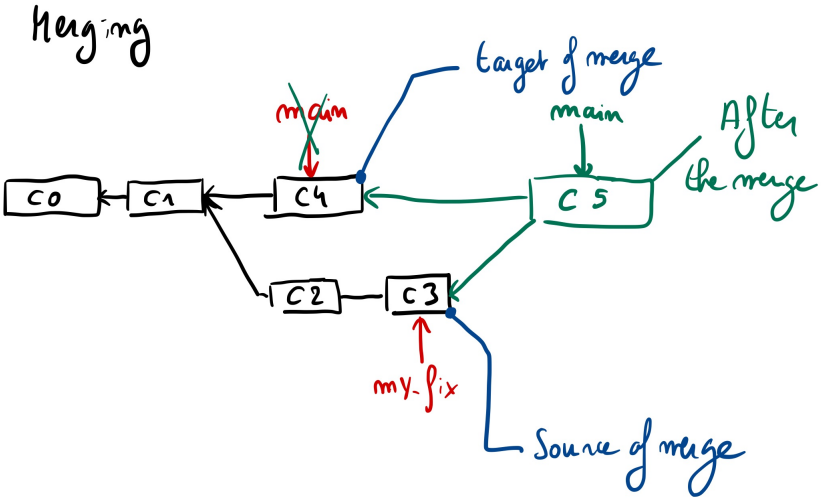
- is asymmetrical (one target, and one source)
- creates a merging commit (having the target and source commit has parents)
- is not destructive (the source branch still exists after ward)

Merge

```
$ git log --graph --oneline --all
    * 6c38de9 (HEAD -> main) Add file1
    | * ed86875 (main2) Add file2
    |/
    * 95129c0 Almost intial commit
```

```
$ git merge main2 -m "Merge commit"
$ git log --graph --oneline --all
    *   39bd944 (HEAD -> main) Merge commit
    |\
    | * ed86875 (main2) Add file2
    * | 6c38de9 Add file1
    |/
    * 95129c0 Almost intial commit
```

HEAD


Merging
target of merge
main
main
After the merge
C0
C1
C4
C5
C2
C3
my_fix
Source of merge

Merge conflicts

Merge conflicts

> *Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging, Git won't be able to merge them cleanly.*
>
> *The Pro Git book*

You have to manually select which changes yout want to keep from each branch and then do the merge commit.

Rebasing

In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase.

The Pro Git book

Merging

- Non-linear history
- Keep all previous commits
- Create a merge commit

Rebasing

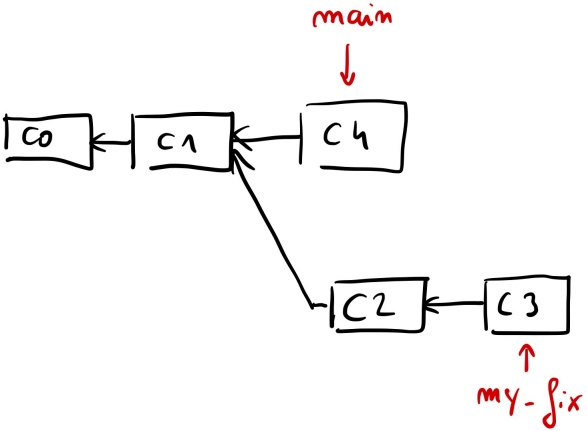
- Linear history
- Create new commits with same modification and delete old commits
- Does not create a merge commit

Before rebase

```
$ git log --graph --oneline --all
    * 3fb7dfb (HEAD -> main) C4
    | * 6008b93 (my_fix) C3
    | * e60ea6c C2
    |/
    * adb048b C1
    * ccee678 C0
```

HEAD

Rebasing 1



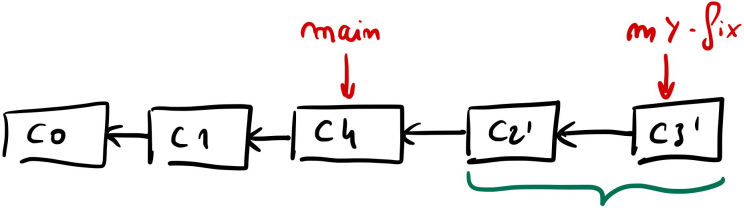
Rebase my_fix on main

```
$ git checkout my_fix
$ git rebase main
$ git log --graph --oneline --all
    * 6efb6ee (my_fix) C3
    * 7bece1a C2
    * 3fb7dfb (HEAD -> main) C4
    * adb048b C1
    * ccee678 C0
```

Linear history, but checksums different.

HEAD

Après rebase de my-fix sur main
Rebasing 2



Different commits objects with same modification

Fast-forward merge

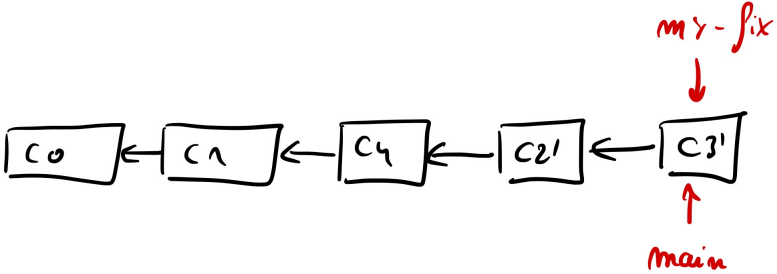
```
$ git checkout main
$ git merge my_fix
    Updating 3fb7dfb..6efb6ee
    Fast-forward
$ git log --graph --oneline --all
    * 6efb6ee (HEAD -> main, my_fix) C3
    * 7bece1a C2
    * 3fb7dfb C4
    * adb048b C1
    * ccee678 C0
```

Fast-foward: merge that does create a merge commit (just moves the pointer, in case of a linear history).

HEAD

Rebasing 3

Fast forward merge de my-fix sur main



Fast forward doesn't create a merge commit.

What if someone started a branch on a commit that was rebased after ?

The Perils of Rebasing

Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:

Do not rebase commits that exist outside your repository and that people may have based work on.

If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family.

The Pro Git book

Revert and blame

Revert

What if someone (including you) made a bad commit ? You can easily correct by reverting a commit.

Revert will create a commit for you that undo the modification.

You should (almost) never just delete a commit and force push it[5]

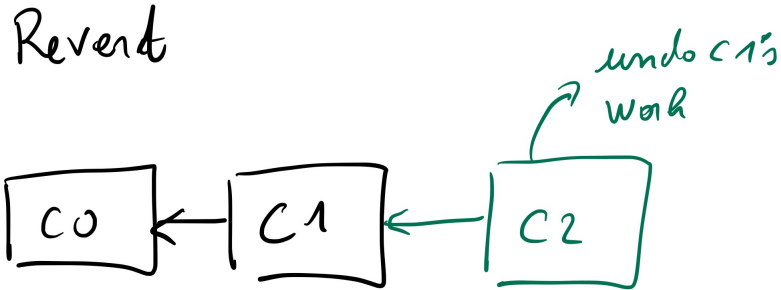
[5] You can delete commits as long as they have not been pushed to remotes.

Revert

```
$ touch C1
$ git add C1
$ git commit -m "C1"
$ ls
    C0  C1
$ git log --graph --oneline --all
    * 76cb4f7 (HEAD -> main) C1
    * 0f3e78a C0
$ git revert 76cb4f7
$ ls
    C0
$ git log --graph --oneline --all
    * fb72e86 (HEAD -> main) Revert "C1"
    * 76cb4f7 C1
    * 0f3e78a C0
```

HEAD

Revert



Who fucked up this code?

Blame

git blame shows what revision and author last modified each line of a file

```
$ git blame alice.py -L 61,63
    f7fafba9 (Yoann Piétri 2023-02-05 14:43:50 +0100 61)
    f7fafba9 (Yoann Piétri 2023-02-05 14:43:50 +0100 62)
    f7fafba9 (Yoann Piétri 2023-02-05 14:43:50 +0100 63)
```

Blame

Local and non-local operations

Remotes

Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you.

The Pro Git book

Remotes

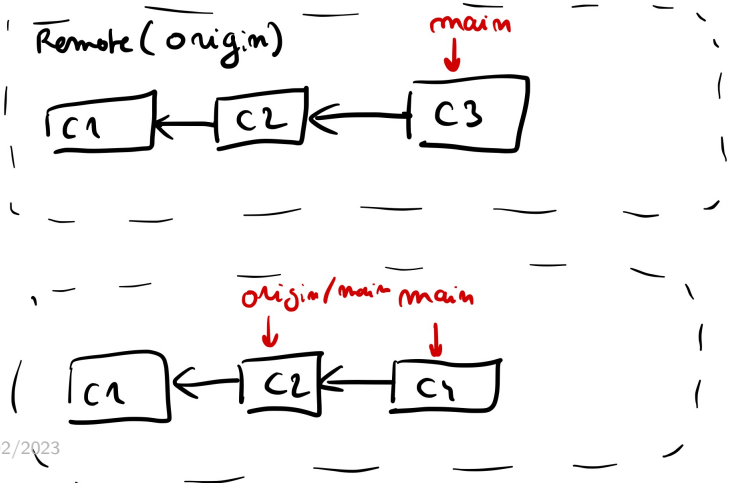
We assume the name of the remote to be origin.

- `git remote show origin`: inspect the remote origin
- `git push`: Push your work to the remote
- `git fetch`: Fetch the new work form the remote
- `git pull`: Fetch the new work and merge it to the local branch

If there is no new commit, `git pull` will only do a fast-forward merge and no merge commit will be created.

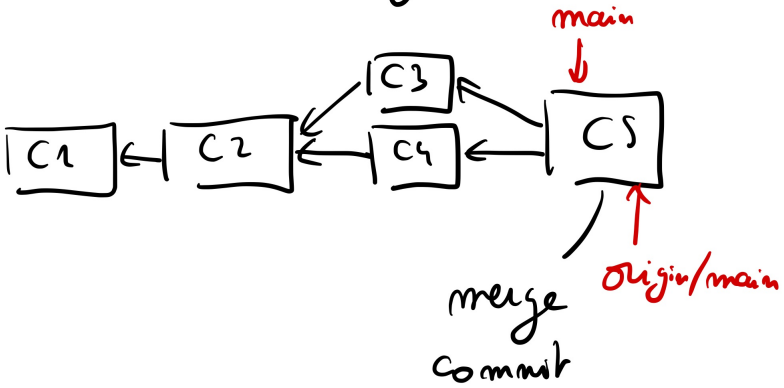
HEAD

Pulling with diverging histories



HEAD

After pull (with merge behavior)



As a rule of thumb, never force push to a remote repository (I see you Paolo)

Force push

Git is local

Most operations in Git need only local files and resources to operate - generally no information is needed from another computer on your network.

To browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you - it simply reads it directly from your local database.

This also means that there is very little you can't do if you're offline or off VPN. If you get on an airplane or a train and want to do a little work, you can commit happily.

The Pro Git book

Nearly Every Operation Is Local

Conclusion

Some cool stuff we didn't talk about

- Reset
- Tags
- Hooks
- Git on the Server
- Signing
- Cherry-picking
- Submodules
- Internals of Git
- Rewriting history (other than rebase)
- Stuff I don't even now about

Conclusion

Git is a free and open source **distributed version control system**.

> *It is also impossible to change any file, date, commit message, or any other data in a Git repository without changing the IDs of everything after it.*
>
> *The Pro Git book*

- Snapshots, Not Differences ✓
- Nearly Every Operation Is Local *(Sorry non-locality)* ✓
- Git Has Integrity *(Not like us)* ✓
- Git Generally Only Adds Data ✓
- The Three States (Working directory, Staging area, Repository) ✓
- Git is cool ✓

The challenge

The challenge - Goal

The goal is to find a string that looks like this

`${CTF}NIPDt9MI0itWJD3FDBvxUBV7RQ9VNa`

(it's start with `${CTF}` and the rest is (almost) random).

You only have to use git commands. You might also use the `cat` command to print the content of a file.

If you need authentification at some point, everything will be already set in place so you don't have to enter any password.

You can now find the useful code here: `https://github.com/nanoy42/git-ctf`

License

Get the source of this presentation here:

`https://github.com/nanoy42/git-presentation`

Origin of memes is written on the images.

Quotes where taken from the Git Pro Book.

The presentation uses the metropolis theme (`https://github.com/matze/mtheme`).

List of useful commands

List of useful commands

- `git commit`
- `git add`
- `git init`
- `git log`
- `git remote`
- `git blame`
- `git push`
- `git pull`
- `git fetch`
- `git merge`
- `git rebase`
- `git revert`
- `git reset`
- `git mv`
- `git rm`
- `git stash`