On the Uses and Benefits of `git`

or how Distributed Version Control Improves Your Life

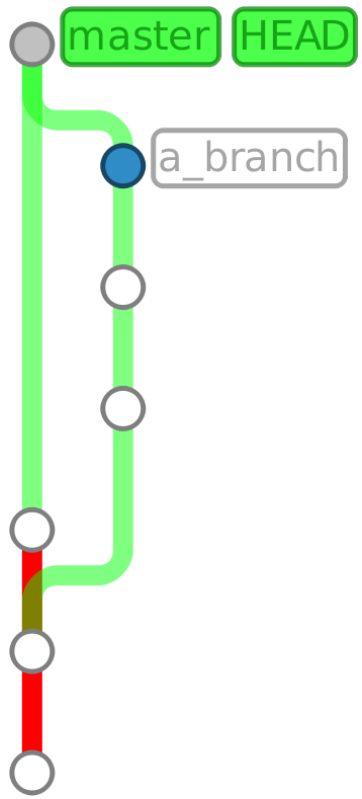
Introduction to Data Science

Christian Holme

2018-02-19

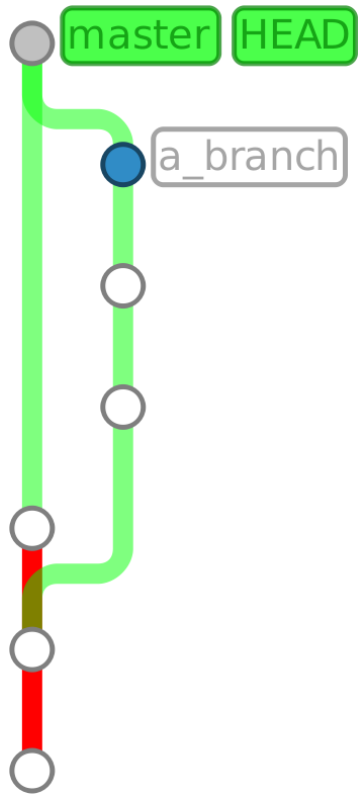
What is `git`?

- `git` is a "version control" system
- It records snapshots of a *repository* (a directory with stuff in it)
- These snapshots are called *commits*
- Each commit records entire state of the repository
- A sequence of commits form a *branch*
- `git` allows (and encourages) branches to diverge…
- …and makes merging easy
- (This is why it is nice for collaboration)



What is `git`?

- `git` is a "*distributed* version control" system
- That means every copy of a repository is equal (in principle)
- Advantage: If you lose your repository, get a copy from somewhere else and nothing is lost!



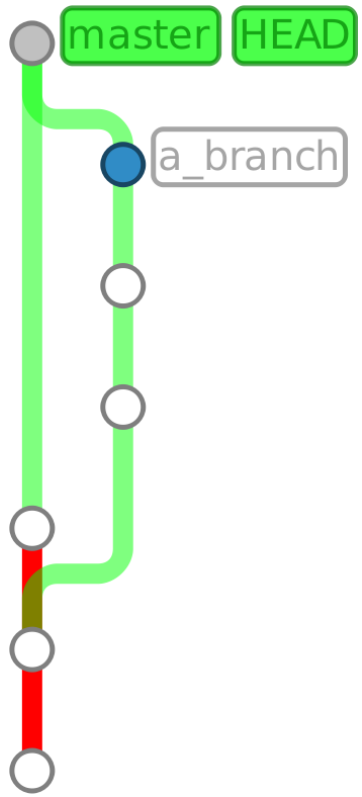
History and git Commits

- git commits are identified by their hash (e.g. de0a35b77d49c27d762cc7aeabc1abc24c159f73)
- git uses SHA1 which has the following properties[1]:
  - deterministic (same input always results in the same hash)
  - quick to compute
  - a small change to the input leads to a large change in the hash (e.g. SHA1("Datascience is cool") = aeb764dcf00d0d302be663e93db5d5a06b9af8b8 but SHA1("Data**S**cience is cool") = 91a990aab48d8c5c396623218b34fbd46ef6845b)
- This enables git to ensure consitency

[1] Partial quote from https://en.wikipedia.org/wiki/Cryptographic_hash_function

History and `git` Commits

- A `git` commit contains:
  - The hash of the *entire* working directory
  - Metadata (commit message, author, date…)
  - The commit-hash of its *parent(s)*
- ⇒ each commit uniquely identifies its parent (which identifies the entire state of the repository), all the way to the initial commit
- ⇒ changing *any* part of the history changes all following commits
- But: For large and changing binary files, a `git` repository will quickly grow (since it stores every snapshot)



Basic `git`

(Example: Basic Commands)

Basic `git`

- Files in a `git` working directory can have three separate states:
  1. Committed files (no changes)
  2. Untracked files or uncommitted changes to tracked files
  3. Files or changes in the *staging area* (These will become part of the next commit)
- In `git status`, these are marked as:
  1. Not shown
  2. Red
  3. Green
- `git`-commands:
  1. `git add <file>` adds the changes to `<file>` to the staging area
  2. `git reset <file>` removes the changes to `<file>` from the staging area (but leaves the file untouched!)
  3. `git checkout -- <file>` discards all changes from `<file>`

A word on commit messages:

Don’t be like Randall Munroe.

Exercise: Basic `git`

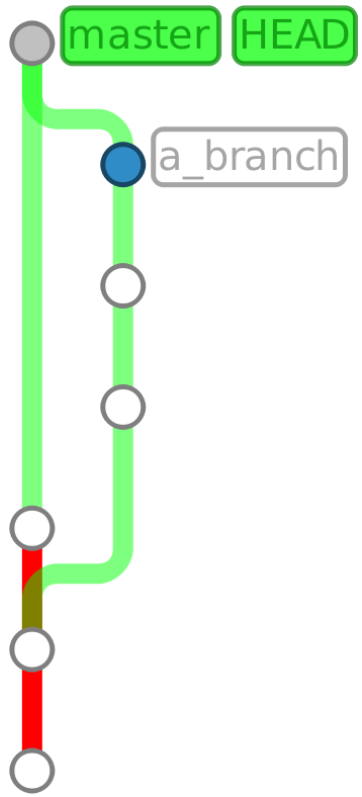
0. (After each commit, run `git status` (it will tell you what is going on). For each command, `git help <command>` can give helpful hints)
1. Create a `git` repository in a folder (`git init`)
2. Create one or more files, `git add` them to the repository and `git commit` the result (remember to write a useful commit message)
3. Change some of the files, show the changes (`git diff`), and commit a subset of the changes to the repository
4. Take a look at the history with `git log` and, if available, `gitk`
5. Then `git commit --amend` your last commit with the remaining changes

`git branch` and `git merge`

- In simple terms: "branching" means multiple commits have a common parent, "merging" means a single commit has multiple (normally 2) parents
- Branching:
  - Create a branch with `git branch <name>` or create-and-switch with `git checkout -b <name>`
  - Work on branch, commit changes
- Merging:
  - Merge <branch> into current branch: `git merge <branch>`
  - `git` is smart about automatically merging unrelated changes
  - But sometimes, we may need to resolve merge conflicts

`git branch` and `git merge`

- A different view: A branch is a series of commits …
- …but since each commit identifies its ancestor (and so on until an initial commit), a branch is also just a pointer to a commit!
- `git commit` moves the pointer of the current branch to the new commit
- `git branch` just creates a new pointer
- `git merge` creates a new commit with two (or more) parents
- (and advances the current branch pointer)



`git branch` and `git merge`

(Example: Branching and Merging)

Working with Remotes

- A *remote* (or *remote repository*) is `git`'s way of referring to another repository
- This is useful when collaborating or as a backup
- Connections can be made through `ssh` or `http` (or just locally)

Working with Remotes

- Important concept: remote branches are remote, we cannot directly interact with them.
- But: `git` has a "local view" of a remote repository via *remote-tracking branches*
- Note: *These are still local branches*
- These are named as `<remote>/<branch>`, so for a branch `master` on remote `origin`, it is `origin/master`

Working with Remotes

- General way of getting changes from a remote repository:
  1. `git fetch` (or `git fetch <remote>`) to update the remote tracking branches
  2. Merge changes into local branch with `git merge <remote>/<branch>`
- This is cumbersome and can be done in one command: `git pull` or `git pull <remote> <branch>`
- This fetches changes from the branch `<branch>` from remote `<remote>` and merges them into the local branch `<branch>`

Working with Remotes

- `git pull <remote> <branch>` fetches changes from `<branch>` on `<remote>` and merges them into the local branch `<branch>`
- Note that:
  a) `<branch>` has to be the same name locally and on the remote
  b) `git pull` is magic and works almost always …
  c) …but when it fails, it is hard to find out way
- Therefore: Even when using `git pull`, think of it as `git fetch` followed by `git merge`

Working with Remotes

(Example: Working with Remotes)

Rewriting History: `git amend/rebase`

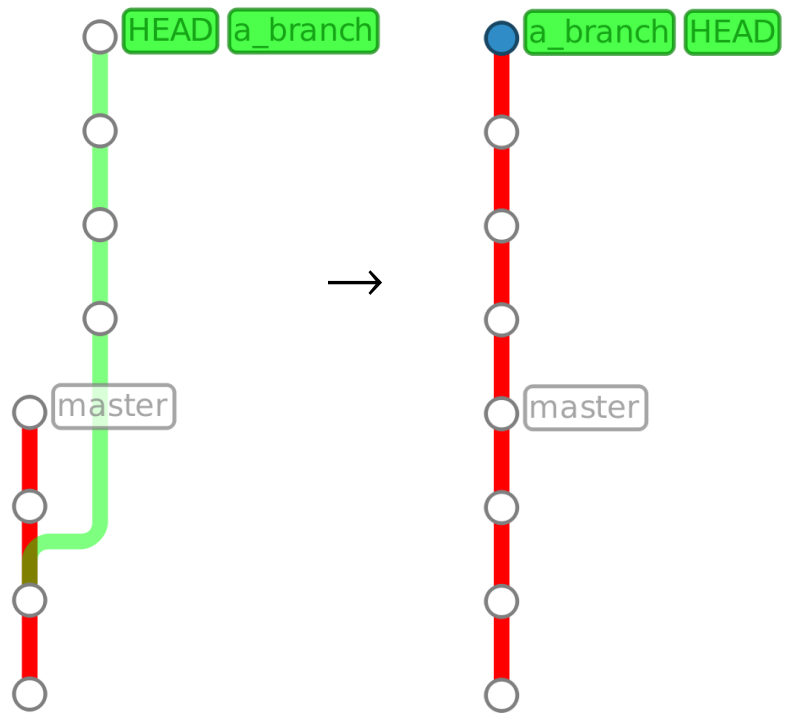
- Normally, `git` ensures a consistent and unchanging history…
- …but it also provides tools to rewrite it if necessary
- A word of advice: Only rewrite the history of purely local changes
- Do not do this if you have already pushed your changes (unless you have to ☺)

Rewriting History: `git amend/rebase`

- We have already seen one way: `git commit --amend`
- Use this when you discover a mistake in what you just committed
- Very useful for correcting typos in commit messages!
- (Why does this rewrite history?)

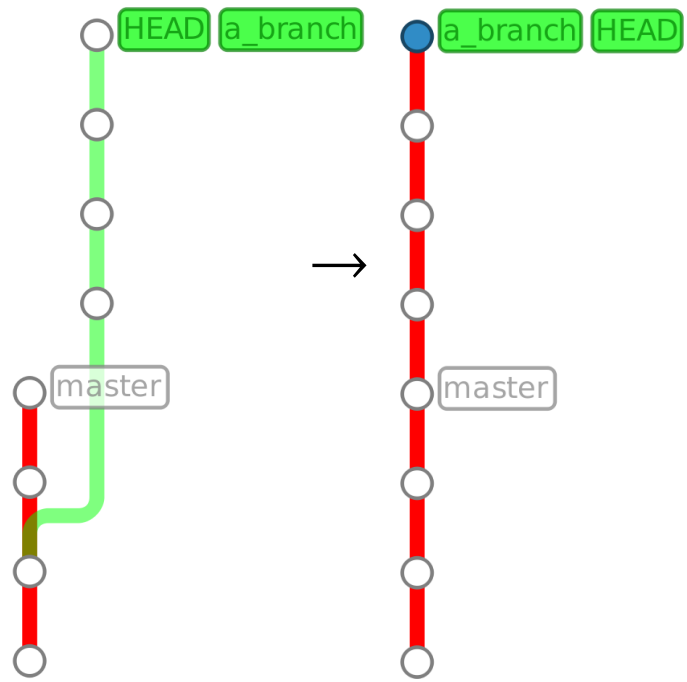
Rewriting History: `git amend/rebase`

- `git rebase <branch>` *rebases* the current branch onto `<branch>`
- Example: It rewrites all commits on `a_branch` as if they had been committed after the last commit on `master`
- Very useful for cleaning complicated histories (since it avoids merge commits)
- For example when working a feature locally
- But: Don't do this to changes that are already pushed



Rewriting History: `git amend/rebase`

- Also useful when incorporating changes from a remote:
- Simply rebase your local changes onto the changes from the remote
- (Use: `git fetch` followed by `git rebase`, or simply use `git pull --rebase`)



Rewriting History: `git amend/rebase`

(Example: `git amend/rebase`)

Rewriting History Part 2: Interactive rebase

- Sometimes, we need to change a commit that is further back…
- …or we want to reorder, remove, or insert commits into the history
- For that, we can use `git rebase -i`: Called interactive rebase
- Regular `git rebase` reapplies commits, one after the other, from some branch onto the current branch
- `git rebase -i` is similar, only we (normally) use it to reapply commits from the current branch …
- …and interactively change them while we do it
- Usage: `git rebase -i <commit>`. We can specify `<commit>` as, e.g. `HEAD~3`, which means: go back 3 commits from the current one

Rewriting History Part 2: Interactive rebase

(Example: Interactive rebase: `git rebase -i`)

Exercise: Rewriting History

1. Go to your toy repository and reorder all commits after the inital commit
2. Then add a new commit just after the inital commit
3. Finally, squash all commits into a single commit

`git bisect`

- Used to find exact commit which introduced a change
- Useful when searching for a bug or regression
- Needs a known "good" and known "bad" commit, then starts a binary search for the offending commit

`git bisect`

(Example: `git bisect`)

Exercise: `git bisect`

1. Clone the `simple-git-repository` at `https://gitlab.gwdg.de/holme/simple-git-repository.git`
2. Use `git bisect` to find the exact commit which added the grid to the graph
3. (Then check the commit message and the changes of the commit you found to see if you are correct)

Jupyter

- This has nothing to do with `git` …
- …but will be very useful for the course:
- At `https://jupyter.gwdg.de/`, the GWDG hosts a service for working with jupyter notebooks
- That is a browser-based tool for creating documents which
  - freely mix code and comments/markup in a presentable way
  - allow easy collaboration

For more information on jupyter: `https://jupyter.org/`

Jupyter

(Example: `https://try.jupyter.org/`)

Jupyter

- We will use jupyter notebooks for teaching this course
- Because they do not separate code from presentation
- Just one caveat: They store results in the notebook
- (e.g. images and plots)
- Therefore, `git` sometimes has a hard time with changes to notebooks

GitLab

- Aaaand back to `git`:
- GitLab is a `git` repository manager
- (similar to GitHub)
- The GWDG maintains an instance that all university students can use
- Offers a useful web interface for collaboration
- Doubles as a backup location for your personal `git` repositories

GitLab

(Example: `https://gitlab.gwdg.de/`)

Exercise: GitLab

Let's clone our Data Science Repository!

1. Go to `https://jupyter.gwdg.de/`
2. Start a new terminal
3. Clone the repository at
   `http://gitlab.gwdg.de/pycnic/datascience-course-ggnb.git`

Links & Sources & Conclusion

- All of this is just a small part of `git`'s power
- Lots of exciting stuff left out (e.g. `git submodules`)

- `https://jupyter.gwdg.de/`
- `https://jupyter.org/`
- `https://gitlab.gwdg.de/`
- `http://gitlab.gwdg.de/pycnic/datascience-course-ggnb.git`
- `git` reference and tutorials:
  - `https://git-scm.com/docs`
  - `https://try.github.io`
  - `https://learngitbranching.js.org/`