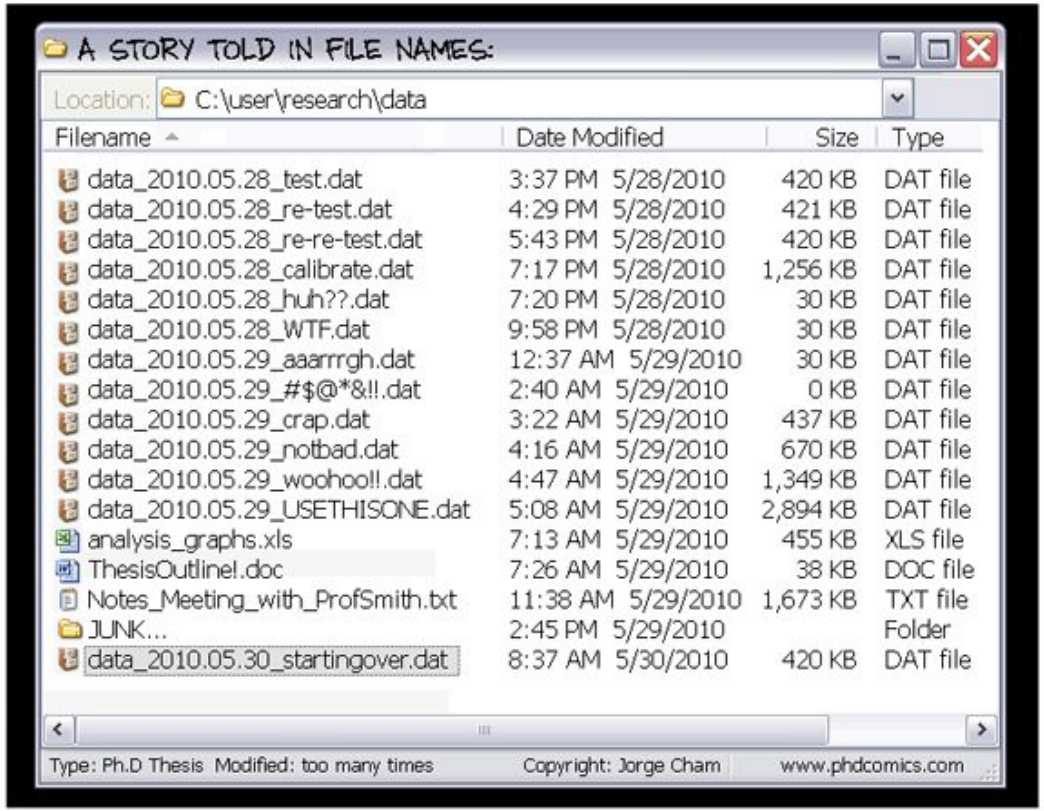
version control with git

short introduction

October 2019 – Jessica Keune

- introduction
  - why version control?
  - git
  - git + github
- git basics
  - add & commit
  - reverts
  - tagging
  - branching
- github
  - sync with github
  - project management
- collaborate
  - clone, push & pull
  - pull requests (github)
  - conflicts
- to advance...
  - further reading



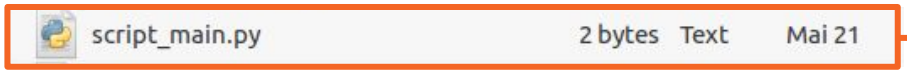
VERSION CONTROL SOFTWARE

"a system that records changes to a file or set of files over time so that you can recall specific versions later"

https://git-scm.com/

- organizes and structures codes/files
- provides a timeline of code/file developments
- can serve as a 'backup'
- enables (updated) code sharing
- facilitates shared code development
- enforces documentation

The code/file becomes **reproducible**.



- initial version
  18-05-2019 17:11
- creating a new function
  19-05-2019 07:47
- calculating indices
  28-05-2019 13:01
- improving index calculation
  28-05-2019 21:19
- incorporating other data
  07-06-2019 08:05
- bugfix: pi is 3.14 now
  14-06-2019 16:59
- adding documentation
  22-06-2019 19:41
- adding plotting functions
  22-06-2019 19:41
- adjust for climatology
  02-09-2019 14:33

git

version control software

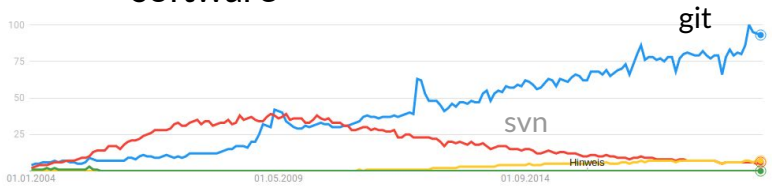
L. Torvald

J. Hamano

features

- <u>software development</u>: tracking changes in code (or any set of files, periodic explicit object packing, cryptographic authentication of history)
- <u>collaboration</u>: coordinating work among programmers (distributed and non-linear development)
- ...

- initial release: April 2005
- creator: Linus Torvald (Finnish-American software engineer)
- maintainer: Junio Hamano (Japanese software engineer @ google)
- current 'standard' in version control software



version control software

features

- <u>software development</u>:
  tracking changes in code
  (or any set of files, periodic explicit object packing, cryptographic authentication of history)
- <u>collaboration</u>:
  coordinating work among programmers
  (distributed and non-linear development)
- ...

management of git repositories
cloud-based hosting service

features

- <u>cloud-based hosting service:</u>
  web-based graphical interface
- <u>project management:</u>
  - documentation (wiki)
  - reporting issues
  - task management
  - ...

<u>largest host of source code in the world</u>
> 100 million repositories (May 2019)
> 28 million public repositories (May 2019)

prerequisites

PREREQUISITE:

you only need to have git installed!

GIT SETTINGS:

to start, let's set some global settings...

```
git config --global user.name username
git config --global user.email email@ugent.be
```

always replace blue text, e.g.,

username → jessica

choose your favorite editor

```
git config --global core.editor "vim"
```

set some nice colors

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

and you're all set and ready to use git!
check your settings with

```
git config --list
```

add & commit

1. change into a directory of your choice
   ```
   cd ../tools/
   ```
2. initialize an empty git repository with name `gitintro` (will create a directory `gitintro`)
   ```
   git init gitintro
   cd gitintro
   ```
3. copy a script of yours or start a simple on (any text file works!)
   ```
   vi program.py
   ```
4. check the status of git in this respository
   ```
   git status
   ```
5. add your program to the repository list
   ```
   git add program.py
   git status
   ```
   → 'staging' (preparing for commit)
6. commit your program to the history (you need to provide a message!)
   ```
   git commit -m "adding my first script to git"
   git status
   ```
7. check your history...
   ```
   git log
   git log --oneline
   ```

continue...
- make some changes to your file, `add` the file and `commit` the changes
- add **another file** to the git repository
- use `git diff` to see unstaged changes

add & commit

- git stores your commit in SHA-1 hash

SHA-1 hash is a one-way mathematical algorithm that maps data of arbitrary size to a bit string of fixed size (*hash value*).

Examples:

fox → ff0f0a8b656f0b44c26933acd2e367b6c1211290
get your ducks in a row → ca58f736ea8a327a2a65a545b499e32fb7c7a0a1

- git stores...
  - the entire content (not just the diff)
  - the commit date
  - the committer's name and email address
  - the log message
  - the id of the previous commit(s)

commit
ff0f0a8b656f0b44c26933acd2e367b6c1211290
c8ae4b58d624680189093ad556e553b41f56bb05
3b9a07eccdc081bf35a826615d1f38f5ace27f72
d85ec0334073fad59f7e29e3d93a54214247b5f3
ca58f736ea8a327a2a65a545b499e32fb7c7a0a1
95ab6a1656430c2e09d2bc28b27f39812eff5c3c
0c0eb81591b8ad4231d79409f2cda981ff43f2b4
c8ae4b58d624680189093ad556e553b41f56bb05
012183fa78f86faa3e480a6ac58e6920b702aa26

add & commit - best practices

- add **every** script (file?) you work on for your paper/phd/project/...
- commit **every** change
  - more commits = more documentation
  - more commits = easier to revert single changes
- use **meaningful commit messages**
  - present tense
  - semantic messages?

```
feat: incorporating other data
|     |
|     +-----> summary
+ -----------> type: chore, docs, feat, fix,
                      style, test, refactor
```

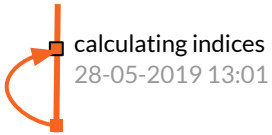
- rule of thumb: you're commit message shouldn't be longer than 50 characters! (or one type?!)
- 'A day without a commit is like a day you haven't worked' (anon.)

https://seesparkbox.com/foundry/semantic_commit_messages
http://karma-runner.github.io/0.10/dev/git-commit-msg.html

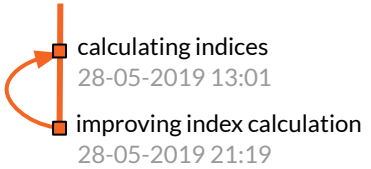
reverts

- or just checkout what you did differently in the past

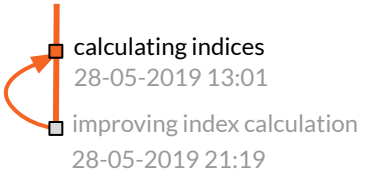
1. uncommited changes (unstaged)



2. commited changes - without history modification (staged)



3. commited changes - with history modification (staged)



reverts (1)

make sure you have *at least 1 commit* in your history

to revert <u>UNCOMMITED (unstaged)</u> changes...

1. change a script / file (but do NOT commit yet)

```
vi program.py
git status
```

2. undo changes (but keep changes in the stash!)

```
git stash
git status
```

3. want to redo the changes?

```
git stash list
git stash apply
```

alternatives to `git stash`:

```
git checkout -- program.py
git reset --hard
```

... but you'll lose your changes.

you can use `stash` multiple times. try also:

```
git stash apply stash@{0}
git stash apply stash@{1}
...
```

reverts (2)

use reverts to document also tests that failed / did not work out.

make sure you have *at least 1 commit* in your history

to revert COMMITED (staged) changes... without modification of the history.

1. change a script / file and commit the changes.

```
vi program.py
git add program.py
git commit -m "I am doing something stupid"
```

2. revert a `commit` / revert to a `commit`

```
git log --oneline
```

option (a): undo a `commit`, but document the undo

```
git revert commit
```

(+ message in your editor)

option (b): just checkout a `commit` (--> staged)

```
git checkout commit program.py
```

option (c): reset to a previous `commit` state (--> unstaged)

```
git reset commit program.py
git reset --hard commit
```

option (b):
use `git checkout -b newbranch` to start developments from here, or simply `add + commit` the unstaged changes again.
use `git checkout -` to return to previous state.
option (c): careful with `git reset --hard commit`

reverts (3)

make sure you have *at least 1 commit* in your history

to revert <u>COMMITED</u> (staged) changes... <u>with modification</u> of the history.

1. change a script / file and commit the changes.

```
vi program.py
git add program.py
git commit -m "I am doing something stupid"
```

2. <u>rebase</u> (modify) your history since `commit`

```
git log --oneline
git rebase -i commit
```

(+ interactive mode in your editor)

the `-i` will open an interactive mode for the rebase, in which you can:

(i) `reword` commit messages,
(ii) `edit` the commit content and message,
(iii) `squash` multiple commits into a single one,
(iv) drop commits (delete line).

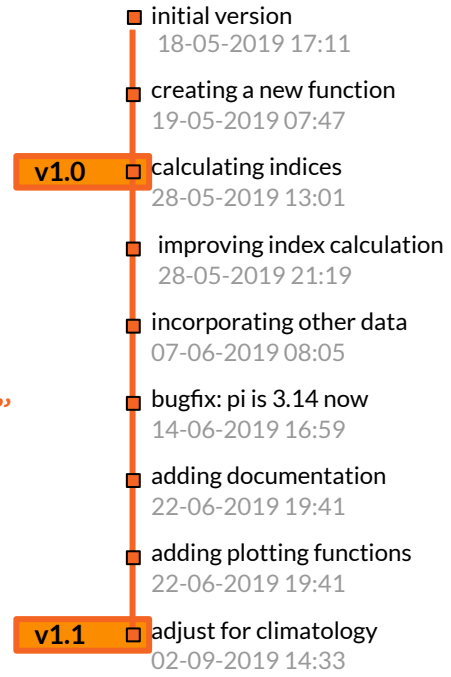
tagging

for paper submission, review, and other 'milestones' ?

to tag specific points in a repository's history as important

1. tag current version using a
   ```
   git tag -a v1.1 -m "submission review"
   git tag
   git show v1.1
   ```
2. checkout a tag version
   ```
   git checkout v1.1
   ```
   (will create a branch named `v1.1`)
3. tag an older version (using the `commit`)
   ```
   git tag -a v1.0 commit -m "first submission"
   ```
4. delete a tag
   ```
   git tag -d v1.0
   ```

further reading...
- `-a` refers to annotated tags (incl. metadata)
- without flags, `git tag v0.2-lw`, creates a lightweighted tag, which is just a pointer to a commit



for stable releases, development and bugfixes ?

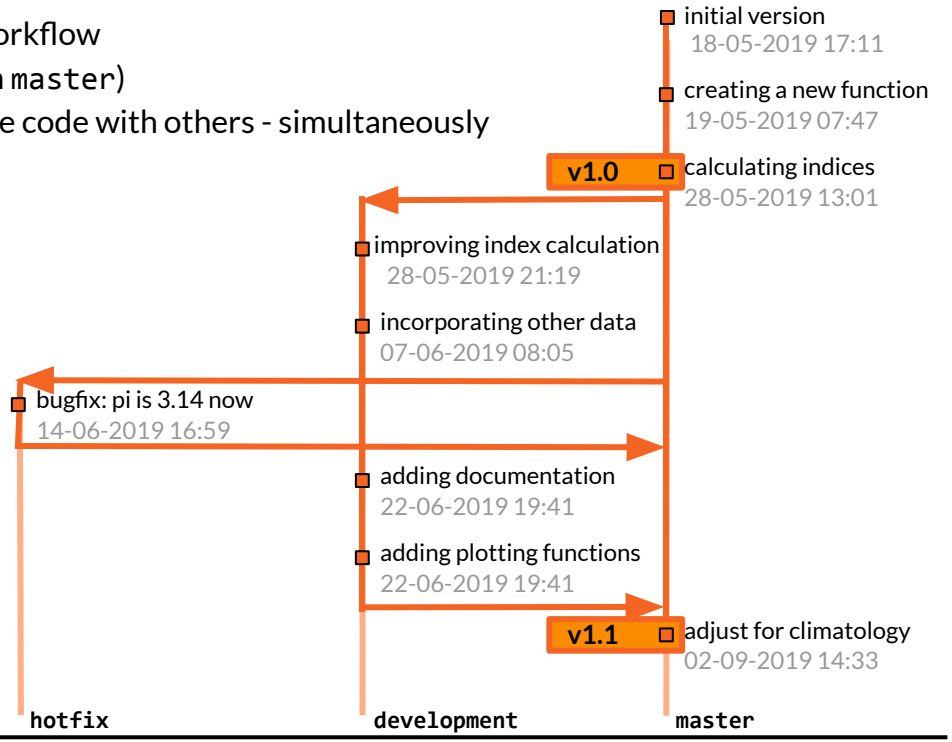
branching

1. check which branch you are working on
   ```
   git branch
   ```
2. make a new branch (starting from *master*)
   ```
   git branch development
   git branch
   ```
3. switch branch
   ```
   git checkout development
   git branch
   ```
4. make changes in this changes, e.g. in `program.py`
   ```
   vi program.py
   git add program.py
   git commit -m “meaningful message”
   ```
5. checkout the difference between your *master* and `development` branch
   ```
   git diff master development
   ```
6. merge your `development` and `master` branch
   ```
   git checkout master
   git merge development
   git status
   ```
7. delete the development branch
   ```
   git branch -d development
   ```

branching

for stable releases, development and bugfixes ?

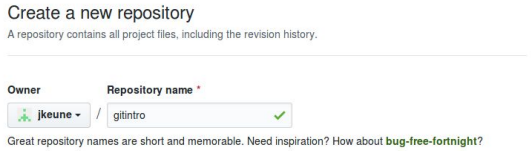
- to organize your workflow
- to share code (from `master`)
- to work on the same code with others - simultaneously



connect your local git repository to a github repository

1. connect to https://github.ugent.be and login with your UGENT username and password

2. initialize an empty github repository



3. add this github respository as a remote URL to your local git (alias `origin`):

```
git remote add origin https://github.ugent.be/jkeune/gitintro
git remote -v
```

4. push your local branch *master* to the remote github repository branch `master`

```
git push origin master:master
```

(enter UGENT username and password)

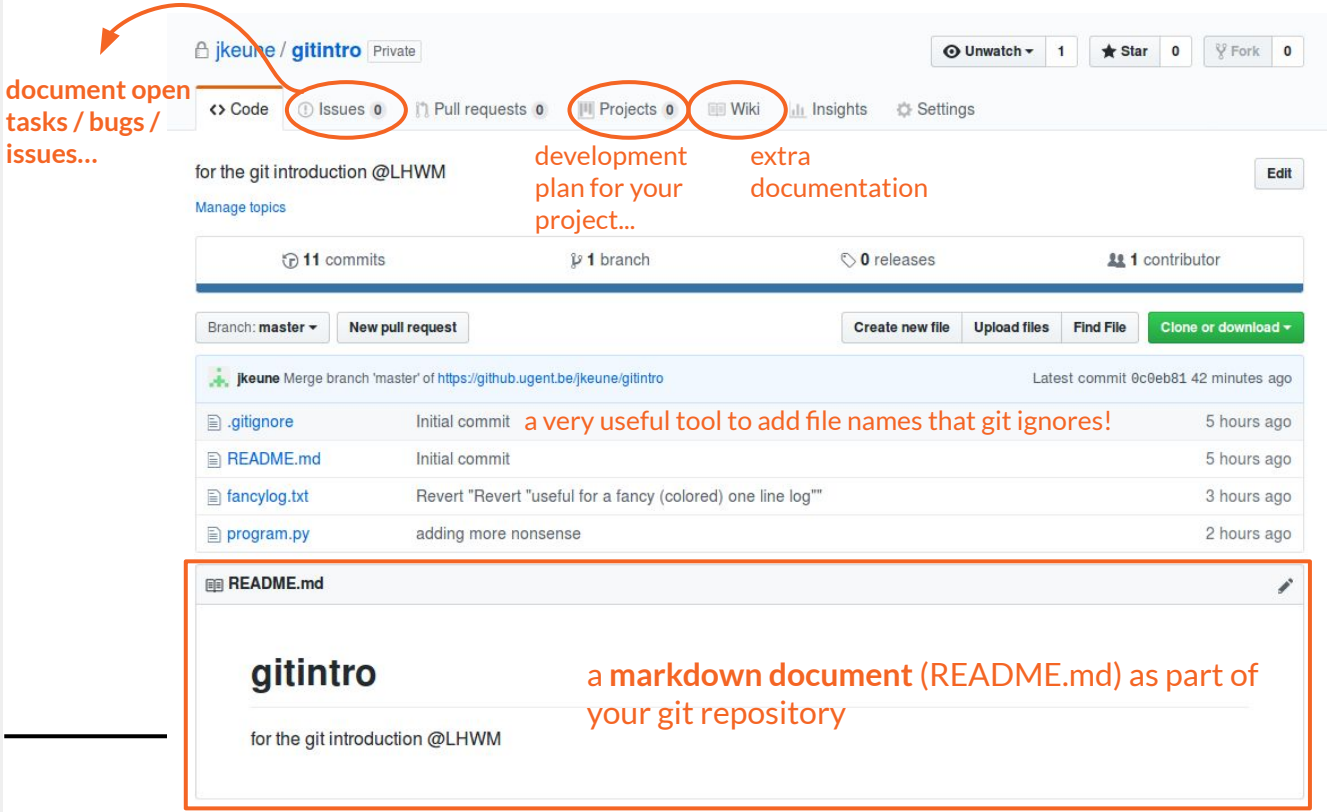
```
git push origin master:master --tags
```

`tags` are not automatically pushed to the remote.

5. checkout the changes on github.

continue...
make some changes to your local git repository and synchronize your changes with github

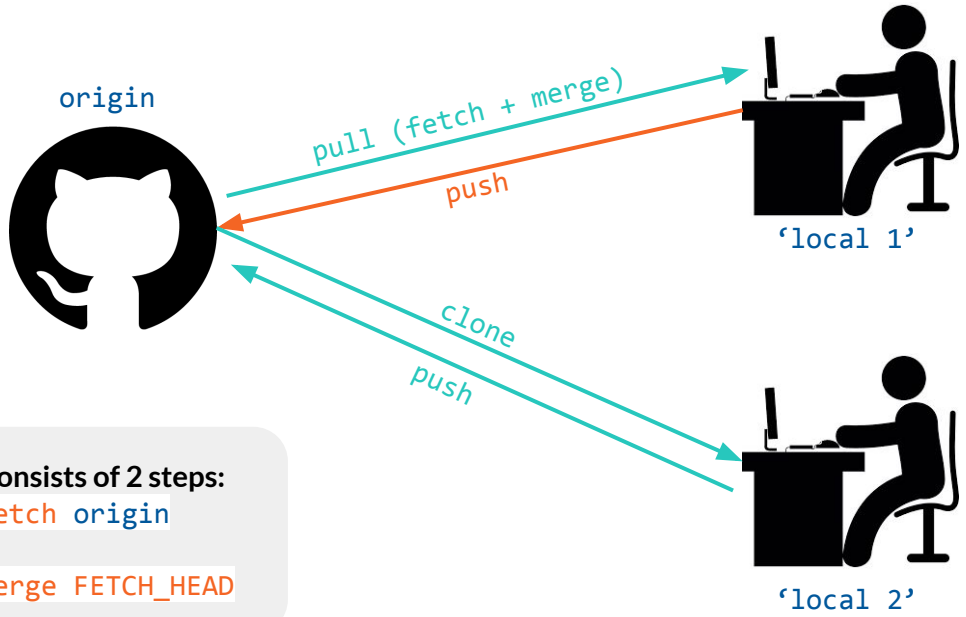
project management on github



what is your github username?

> to continue, please tell me your github username

clone, push & pull



`pull` **consists of 2 steps:**
`git fetch origin`
and
`git merge FETCH_HEAD`

clone, push & pull

1. change into a path without a git repository, e.g.
   ```
   cd tools
   ```
2. clone the branch https://github.ugent.be/jkeune/gitdemo
   ```
   git init
   git clone https://github.ugent.be/jkeune/gitdemo
   git remote add origin https://github.ugent.be/jkeune/gitdemo
   cd gitdemo
   ```
3. make a branch with your name (e.g. jessica) from `master`
   ```
   git checkout -b name
   ```
4. make changes to a file in that branch (choose any you find!)
   ```
   vi file
   git add file
   git commit -m "feat: introducing a new feature"
   ```
5. push your changes to the branch origin
   ```
   git push origin name
   ```

pull requests

1. go to
   https://github.ugent.be/jkeune/gitdemo
   and make a `pull request` from your branch `name` to another branch `name2`
   (e.g. pull request from `jessica` to `femke`)
2. assign someone else to review your changes / pull request
3. let this person approve the changes
4. before you merge them into the `name2` branch

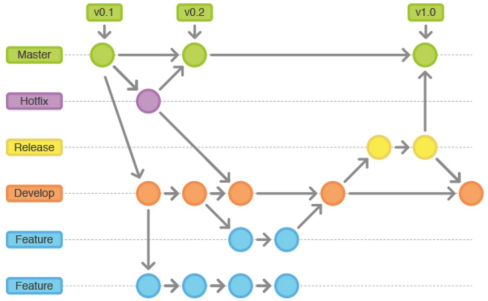
conflicts

* i will modify one file, which you will modify as well (simultaneously) and you will try to resolve the conflict that emerges when pushing/pulling.

1. Change to the master branch of gitdemo
   ```
   git checkout master
   ```
2. Modify a file in `master and add and commit the changes`
   ```
   vi file
   git add file
   git commit -m “meaningful message”
   ```
3. Try to pull the remote `master` branch
   ```
   git pull origin master:master
   ```
4. Fetch the remote `master` branch and try to merge
   ```
   git fetch origin
   git merge FETCH_HEAD
   ```
5. Try to solve the conflict:
   a. search for these lines in the file … and delete everything you don’t need
   b. leave the editor and save
   ```
   git status
   git add file
   git commit -m “resolving merge issues”
   ```

```
<<<<<<< HEAD
new message
=======
old message
>>>>>>> branch
```

further reading...



Tutorials
https://backlog.com/git-tutorial/

Cheat sheet:
https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf

Understanding how git works:
https://git-scm.com/book/en/v2/Git-Internals-Plumbing-and-Porcelain
https://hackernoon.com/https-medium-com-zspajich-understanding-git-data-model-95eb16cc99f5

Add on's:
git-flow: https://danielkummer.github.io/git-flow-cheatsheet/

... and `google`!