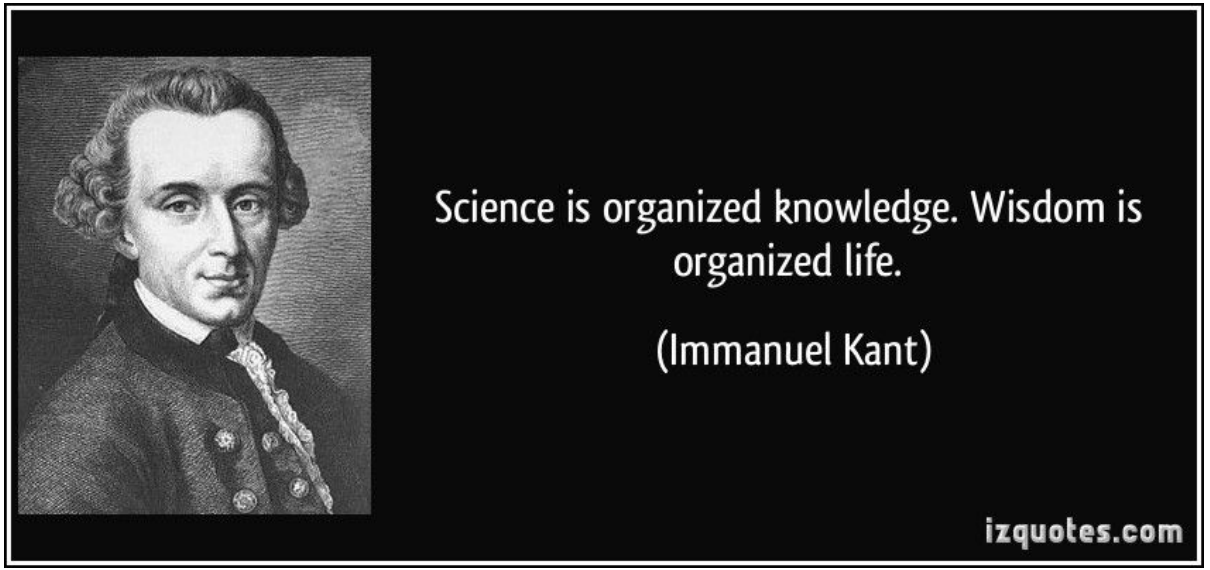
A crash course in Version Control

2020-04-03

Burcu & Paul

for infektiologie.usz.ch



Theory

Version Control

- the management of changes to **collections of information** over time
  - collection of information (here) = a tree of folders containing text files (code)
- developers could simply retain multiple copies of the different versions of the program, and label them appropriately
- Version Control **System**: **enforces the discipline** of keeping all versions around and well labeled (author, date, reason for change) and **as a reward supports you** in all tasks that have to do with versions
  - compare
  - merge
  - share
  - collaborate, review other’s work
  - backup, archive

Variants

- we might just keep a **chain** of versions (revisions) of one single “**master**” **variant** of our code (tutorial part 1)
  - this is what Wikipedia does for its articles and maybe (hopefully…) your file system and cloud storage does it for your files
- it gets more interesting when we can create variants or copies that multiple people can work on independently simultaneously and when both have finished their additions we **merge** them
- in version control, we usually call different variants **branches**
- (if variants are not intended to be merged again, they are called forks instead)
- variants might be:
  - future main variant
  - experiments
  - unfinished work

Version Control System

- definite way to track and account for changes to documents
- helps you, **if** you **use it properly**
  - **tell it** which revisions your current revision is based on (instead of manually copying the changes)
  - apply best practices when using it: commit often, …, etc.
- tasks of the Version Control System:
  - protocol changes (who changed what when)
  - restore any old state
  - archive and backup a complete history in a well known format
  - coordinate shared access to files
  - allow simultaneous editing of different variants

Version Control Software

- implements the Version Control System
- on your computer: usually comes with different interfaces
  - command line interface
  - graphical user interface
  - plugin for your IDE (integrated development environment) such as RStudio
- on a server
  - optional
  - **remote** shared state of repository, just stored on disk (.git folder)
  - **origin** of everyone's local copy
  - graphical website to access it
  - API to modify remotely
  - cloud services (SaaS, software as a service): github, bitbucket, ...
  - self-hosted: gitlab, bitbucket, …
  - serves as backup & archive

Why should I care about Version Control

- a data scientist wants to answer questions about data
- he cannot if he loses the data
- questions are expressed in code
- code is data
- we can afford to never delete any code you can type in over your entire life
  - don’t permanently overwrite (delete) things, you might want to go back...
- scientists can reproduce their results
  - computers are deterministic
  - data shall be unchangeable
- they know what their program looked like when it created some results
  - whether they were right or wrong
- the historical record of information can and should be complete
  - it is the only thing we can preserve without flaw forever...

git Theory

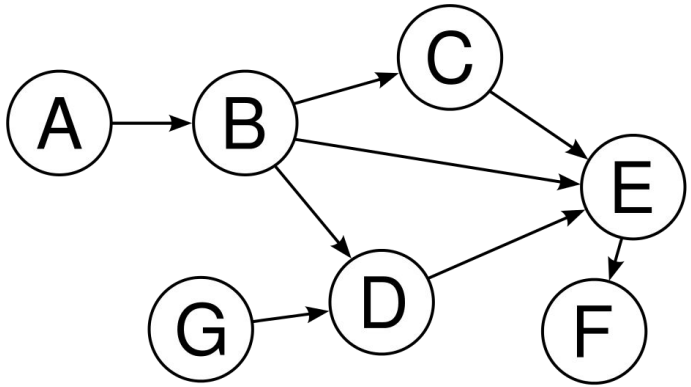
Vocabulary

You have to learn this to understand git's output and documentation and use its commands.

- **commit** (revision, version), snapshot: the complete **unchangeable** collection of information at some point in time
- **repository**: database of all revisions plus labels and comments (commit messages) and dates (metadata)
- **working copy** (**working tree**): a copy of a revision extended with your latest uncommitted modifications
- **clone**: copy a repository *while keeping a remote link to origin...*
- **checkout**: make a **working copy** from a specific revision
- **index, staging area, staged (added) changes**: changes for next commit

The git Model of Variants

- a directed acyclic graph of revisions (commits) (polyhirarchy)
- every node builds on the revisions pointing to it
- analogy: scientific paper citations and influence
- forking/branching
  - creating branches
- joining
  - merging
  - pull request, aka. merge request
  - after merging C and D into E, all commits that went into C or D are part of the history (log) of E



Commit Hashes, Tags, Branches, HEAD

- every revision (every commit) is labeled with an essentially globally unique commit hash
  - f4cd71ad633f1ce888448560be82285271c9f541
- we can define aliases (names) for commits using **tags** and **branches**
- branches always move along with **us** when we make a new commit while tags stay behind
- HEAD is a pointer to the commit, tag or branch that the current working copy was created from
  - if it is not pointing to a branch, we are in “detached HEAD” mode
  - it locates **us**

Giving Names to Nodes: Commit Hashes, Branches, Tags

- you don’t choose commit hashes
- tags are often used to annotate specific released versions: “v1.0.0”
- common branch names:
  - **master** - usually the latest **known good** releaseable state
  - **develop** - the result of merging all recent changes which should work together
  - feature/speed-up-processing - a common convention for names of branches with new features
  - bugfix/should-tolerate-empty-files - branches with bugfixes
  - *recommendation: don’t store important information in branch names and clean them up*
- a branching strategy, aka. branching model should be agreed upon by collaborators to have consistent names
  - e.g. https://nvie.com/posts/a-successful-git-branching-model/ or https://guides.github.com/introduction/flow/ ...

Synchronizing Repositories

git keeps it simple: Just copy all of the .git folder somewhere

→ Synchronize to github or gitlab through https or ssh

Practice

git tutorial

Technical Prerequisites

- git is installed and configured
  - `$ git --version`
  - `git version 2.17.1 (OR SIMILAR)`
  - `$ git config --global user.name`
  - `$ git config --global user.email`
- you have a github.com account that you are logged in to
  - such as https://github.com/Masterxilo
- for ssh setup: you have a local ssh id_rsa(.pub) key pair
  - `$ cat ~/.ssh/id_rsa.pub`
  - if not:
  - `$ ssh-keygen`
  - (and/or for https: `$ git config --global credential.helper store`)

Part 1 - Starting Alone from Scratch

Chat: https://app.slack.com/client/TUYAKERK5/C0105N30CMT

Create some information

```
$ cd
```

Same as `cd ~` or `cd $HOME`

```
$ mkdir my-git-project && cd my-git-project
```

```
$ echo "Hello World!" > hello.txt
```

```
$ cat hello.txt
```

```
$ ls -a
```

No git here so far

Turn this into a git repository

```
$ git init

Initialized empty Git repository in /home/dev/my-git-project/.git/
```

Make this folder a git repository. Creates hidden .git repository folder. Creates & checks out (points HEAD to) branch “master”. There are no commits yet.

```
$ ls -a

$ git status

On branch master
```

Our file is currently **untracked**: git will not care about its changes and others will not see it. This can be useful for secret and big files (--> .gitignore, later…)

Add our first file

```
$ git add hello.txt
```

Add our file to index/changes to be committed/staging area/”cached” files (“stage” the file).

```
$ git status
```

Show what is in the index (staging area), what will be commited next.

```
$ git commit -m 'put any commit message here!'
```

Modify the information

Use any tool you want, nano, gedit, vscode, RStudio...

```
$ nano hello.txt
```

Create more stuff

```
mkdir -p ./sub/folder && echo "more" > ./sub/folder/stuff.txt
```

When you have madea few changes.

Ask git what we changed so far

```
$ git status
```

```
$ git diff
```

Add all untracked files to next commit

```
$ git add .
```

Make sure git now considers all relevant files as Changes to be committed next:

```
$ git status
```

Then commit

```
$ git commit -m 'explain what you changed and why'
```

Explore the repository you just created

```
$ git status

nothing to commit, working tree clean
```

Great. We have no changes (and no untracked files) so we can go somewhere else in the history.

Git still remembers all revisions for us (stored in .git)

```
$ git log
```

Copy a specific version to working tree (check out)

```
$ git checkout b20258df42e88c37b77f4363a859f
```

```
$ ls -R
$ git status
$ cat hello.txt
$ git checkout master
```

We switched back again to “latest commit on master branch” (= the commit currently labeled “master”).

```
$ cat hello.txt
$ ls -R
```

```
Change more and commit
```

```
$ echo a > a.txt && git add . && git commit -m 'added a'
```

```
$ git log
```

Restore single or all files to version in latest commit

```
$ rm hello.txt
$ # oops
$ git checkout -- hello.txt
$ # uff

$ rm -rf *
$ # ohoh
$ git checkout -- .
$ # phew. Note: . usually means “all” in git
```

As long as .git is preserved, all **commited** data is safe! Commit often!

Intermezzo: remote git access with ssh & https

Encrypted & authenticated & authorized transmission via ssh

- to copy your repository to a remote server securely (or to clone it from there)
- encrypted: no one can intercept your transmission and see what you are sending
- authenticated: you authenticate/identify yourself to the server so that he knows who you are
  - proof of identity is confirmed by ownership (knowledge) of private key corresponding to **the public key that you gave to the server → next slide**
- authorized: the server can define what your identity (user account) may do to the repository
  - many git services implement some form of (per-repository & per-branch) permissions...

How to give your public key to github

https://github.com/settings/security

Alternative: transmission via https & Basic Auth

- encrypted: no one can intercept your transmission and see what you are sending
- authenticated: you authenticate/identify yourself to the server so that it knows who you are
  - proof by knowledge of username:password combination that the server can verify
- authorized: the server can define what your identity may do to the repository

Let git store https Basic Auth credentials

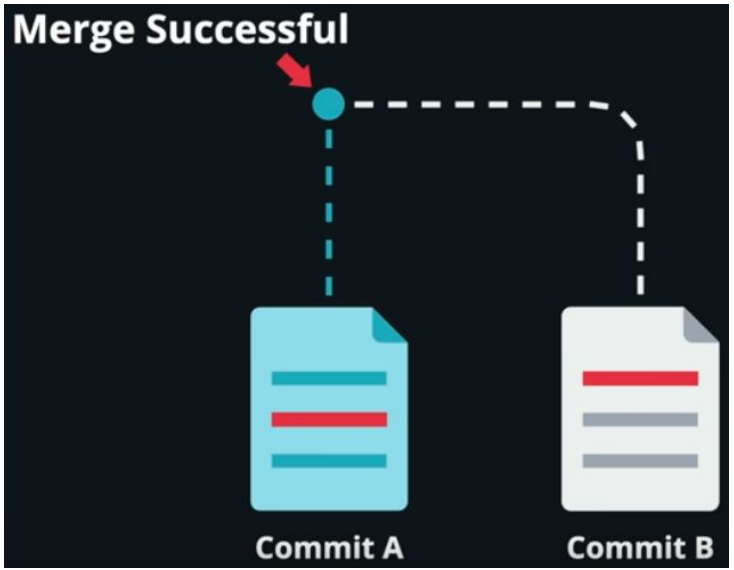
```
$ git config --global credential.helper store
```

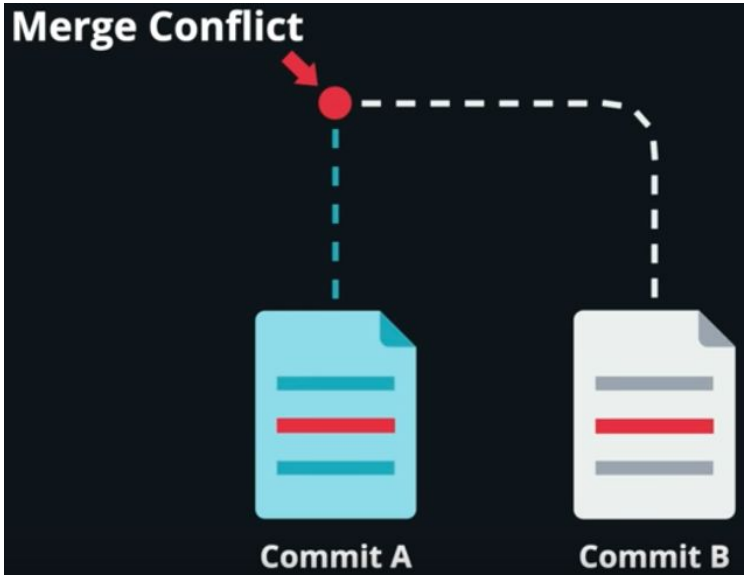
Part 2 - Working in Teams

Pairings: https://docs.google.com/spreadsheets/d/15zfHnnh1yVN3bZvlFncOMIgTf46D5OuN79lgt8DnKnU/edit#gid=0

When does a conflict happen?

You and your team member changed the **exact** same line of code / text independently...





git clone to add a remote

Resources
Outlook: Advanced Topics

Cheat Sheet git Commands

I use (only) these every day

- (git init)
- git clone
- git push
- git pull
- git checkout <branch>
- git checkout -b <new-branch-name>
- git tag
- git commit -m 'commit message'
- git add -A OR git add . OR git add ./folder/file.txt
- git merge
- (git)rm
- (git)mv

Documentation: https://git-scm.com/docs or `man git` or `git add --help`

What to learn next

- maybe you work more efficiently with a graphical git interface - there are many: Source Tree, gitkraken, Github Desktop
- .gitignore
  - ignore your huge data files, private/secret information and processed output
  - use other means and channels to share them if necessary
  - don’t version-control computable/derived information unless you have a good reason to
  - good practice: treat input and output files as immutable, always use new names
- disciplines
  - not all possible ways of using git are sensible, widely adopted or proven to be effective
  - branching models
    - how to name your branches
  - commit
  - linking code and commits to issues in an issue tracking system
- https://en.wikipedia.org/wiki/Version_control https://git-scm.com/book/en/v2

Aside - Version Control in NGSDB

Every change to the db is a commit.

Currently, it only implements a “one branch/version chain per data item” strategy, no forking/branching/merging.