&

GitHub

Workshop

How we learned to love the git

Who are we?

Assistant Professor, Research, Health Data Science (HeaDS).

Data Scientists, SUND Data Lab, Centre for Health Data Science (HeaDS).

Post-doc, Computational Biology Laboratory, Danish cancer society Research Centre (DCRC).

Jonas Sibbesen

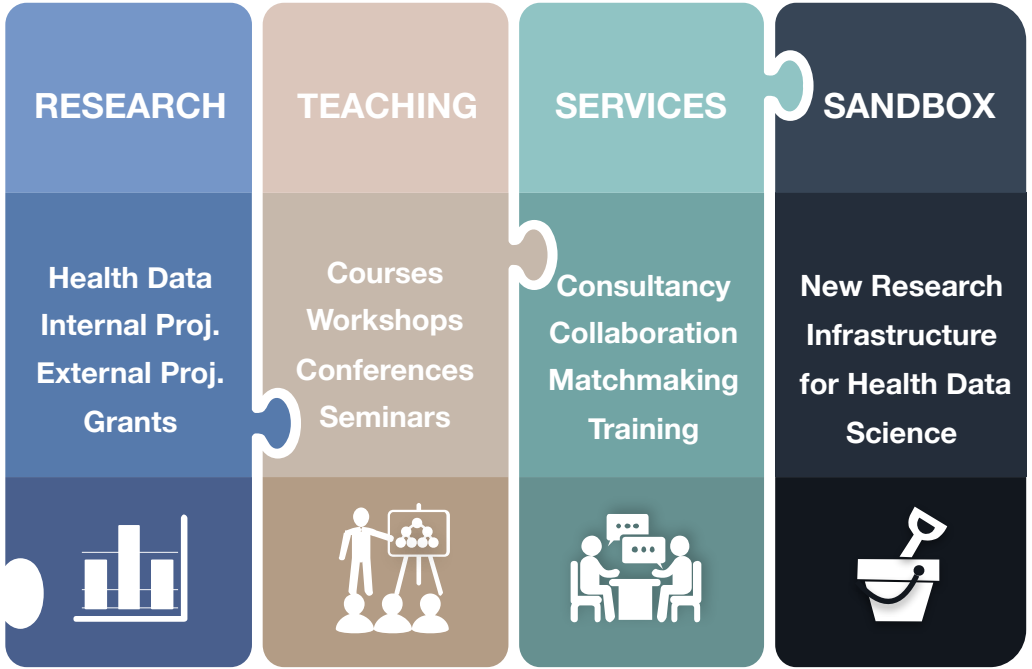
Diana Andrejeva

Henrike Zschach

Matteo Tiberti

Center for Health Data Science

- **Conduct Health Data Science research.**
- **Serve as the UCPH Data Lab.**
  - Be the hub for health data science research.
  - Provide SUND researchers with health data science services & training.
- **Develop National Health Data Science Sandbox for Training and Research**



https://heads.ku.dk/

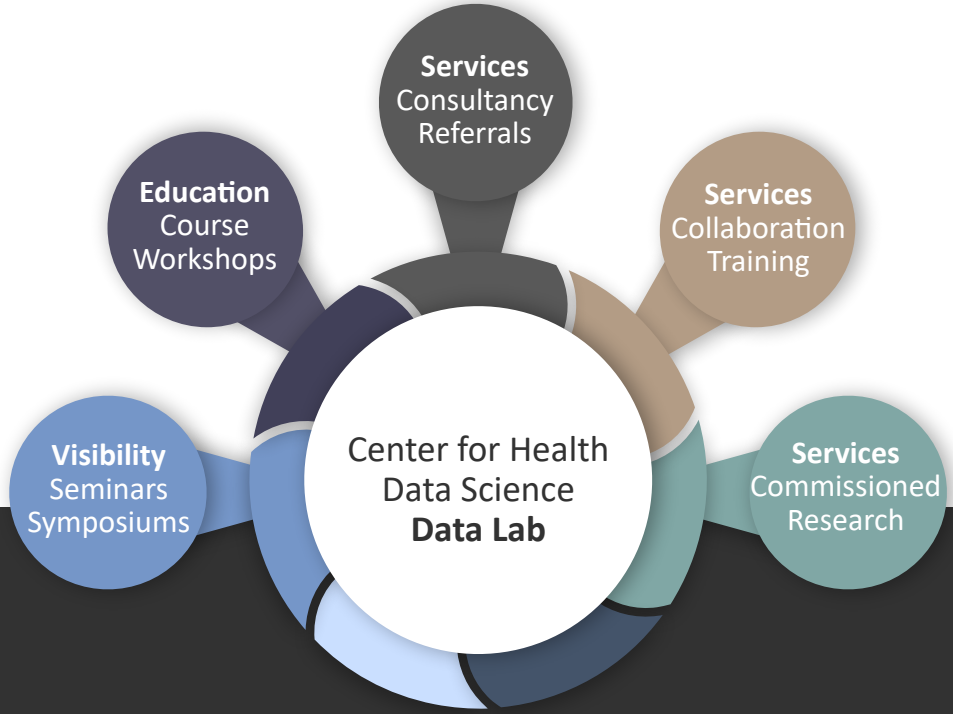
SUND DATA LAB


Diana Andrejeva


Henrike Zschach


Eleonora Nigro

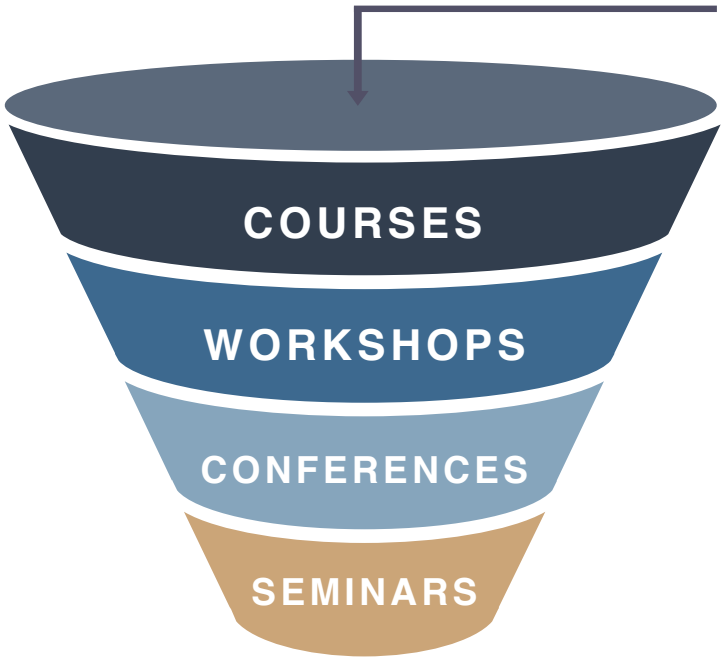

Thilde Terkelsen



Website - https://heads.ku.dk
Email - datalab@sund.ku.dk

Follow us on **Twitter** at **@ucph_heads**!
Slack channel and **mailing list**
courses, conferences, research activities, job postings, etc. within the field of data science.

Center for Health Data Science



git/Github Workshop (Reproducible Science): Today.

From Excel to R (module 1): March & June, 2022. More R modules in collaboration bio-statistics.

Python Tsunami: May.

Computerome 2.0 Users Workshop:
May, 2022.
Collaboration Globe.

Introduction to RNA-seq analysis, June 2022.

Program

9.00-9.45: Theoretical introduction
9.45-9.55: First look at git and github, first commands
9.55-10.15: Ex 1-3: GitHub repository, local copy, status

10.15-10.25: Staging, committing, checking history
10.25-10.40: Coffee Break
10.40-11.00: Ex 4-5: First commits

11.00-11.10: Checkout and fix commits
11.10-11.30: Ex 6: Checkout, reset, revert

11.30-11.40: Branching and merging
11.40-12.00: Ex 7: Branching and merging

12.00-13.00: Lunch break

13.00-13.15: Q&A

13.15-13.25: Merging conflicts
13.25-13.45: Ex 8: Creating a conflict and resolving it

13.45-13.55: Interacting with a remote (pull, fetch and push)
13.55-14.15: Ex 9: Interacting with a remote

14.15-14.30 Coffee Break

14.30-15.00: Ex 10: Contributing to our shared recipe repository - Issues and pull requests

15:00-15:30: Q&A (and working on a repository for your own code)

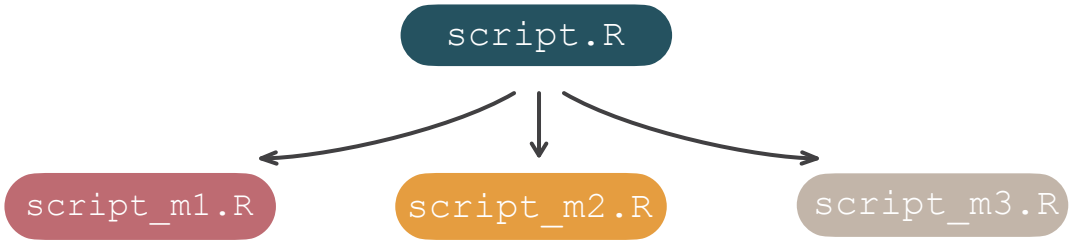
Resources

- Git Pro Book: https://git-scm.com/book/en/v2
- Introduction to git for bioinformaticians: http://ponderomatics.com/git.html
- Get started with SourceTree: https://confluence.atlassian.com/get-started-with-sourcetree
- CodeRefinery website: https://coderefinery.org/lessons/

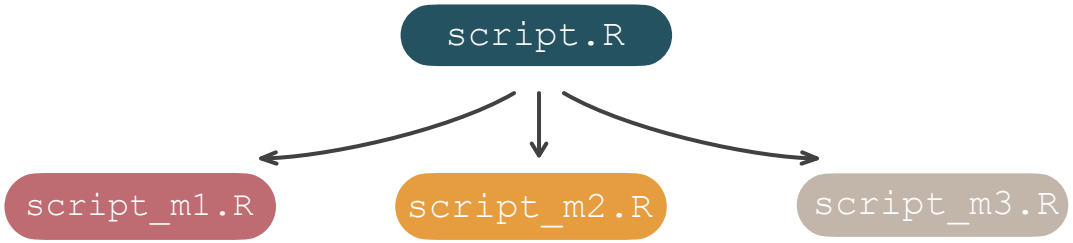
Justification

```
script.R
```

Justification

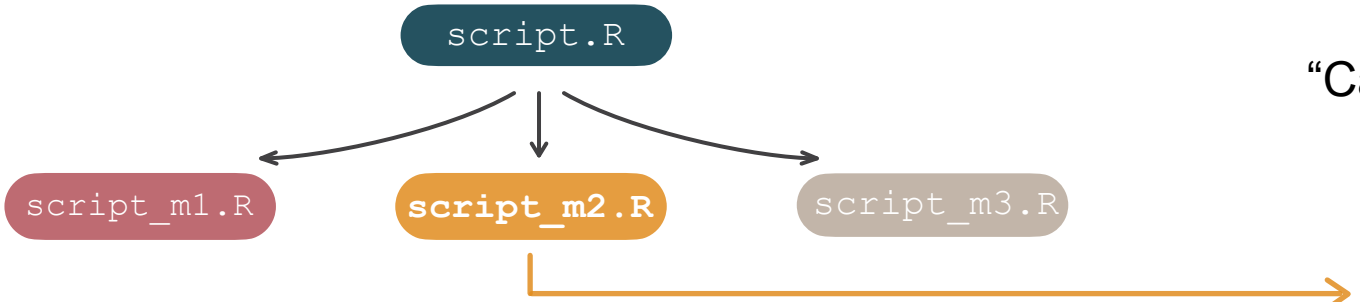


Justification



README

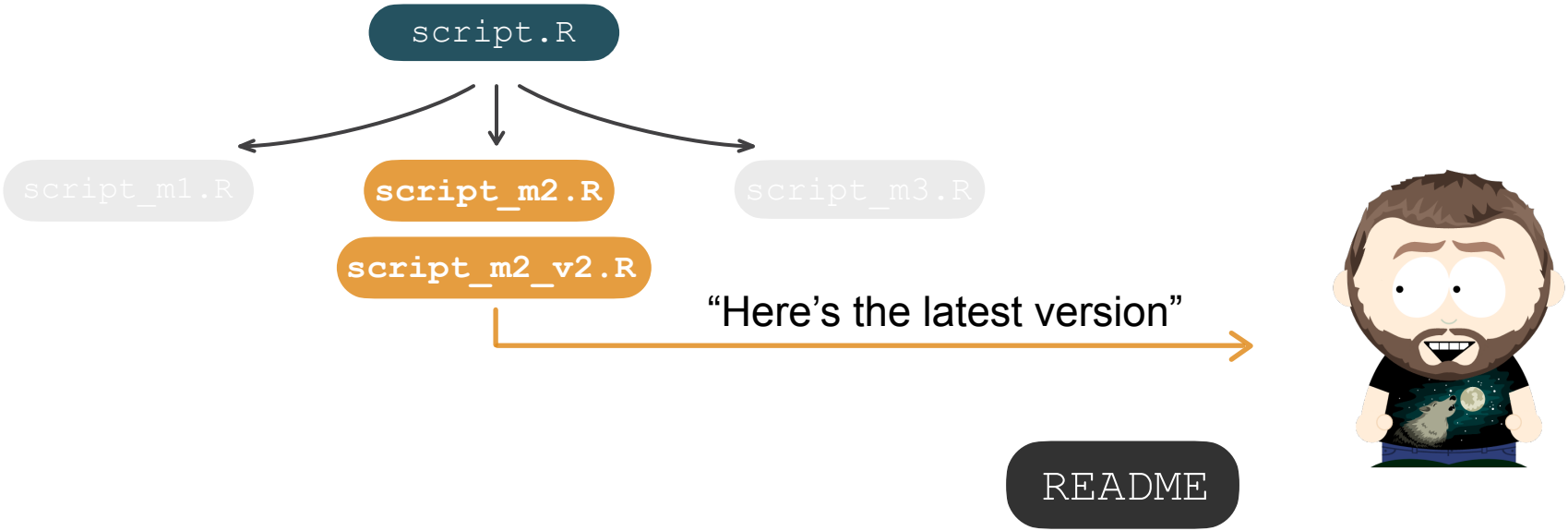
Justification



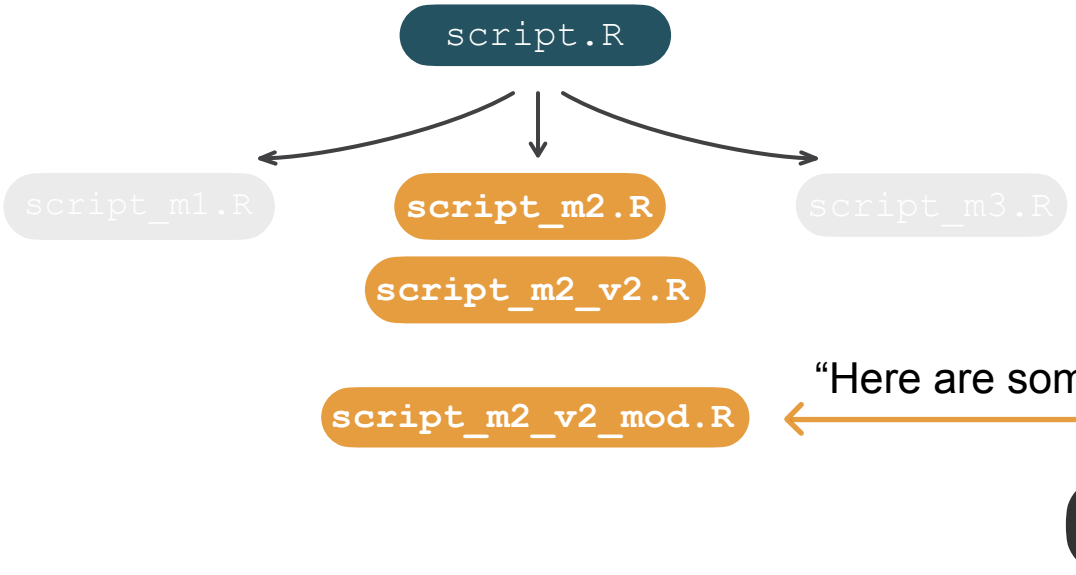
“Can I give it a try?”

README

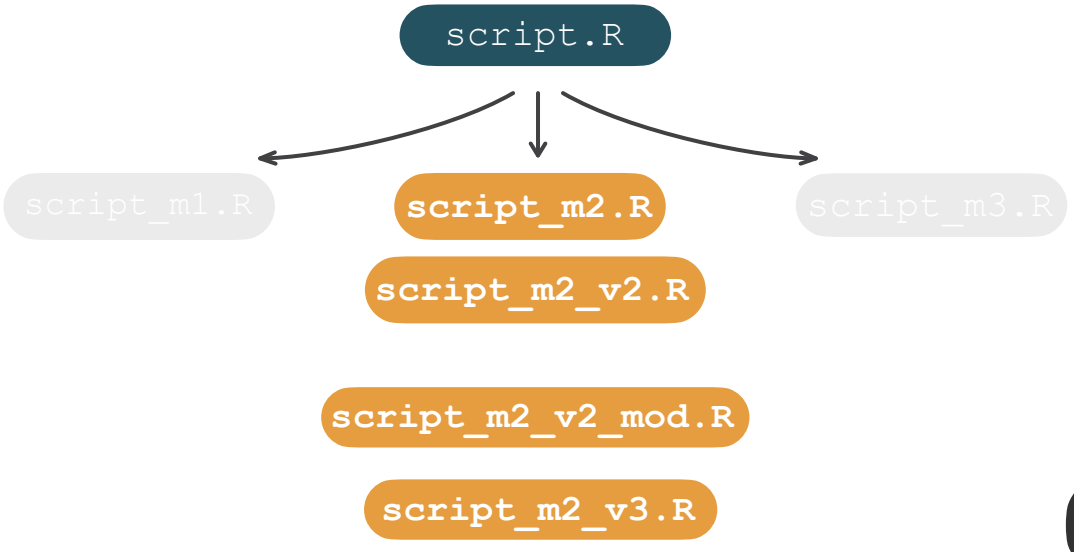
Justification



Justification



Justification



README

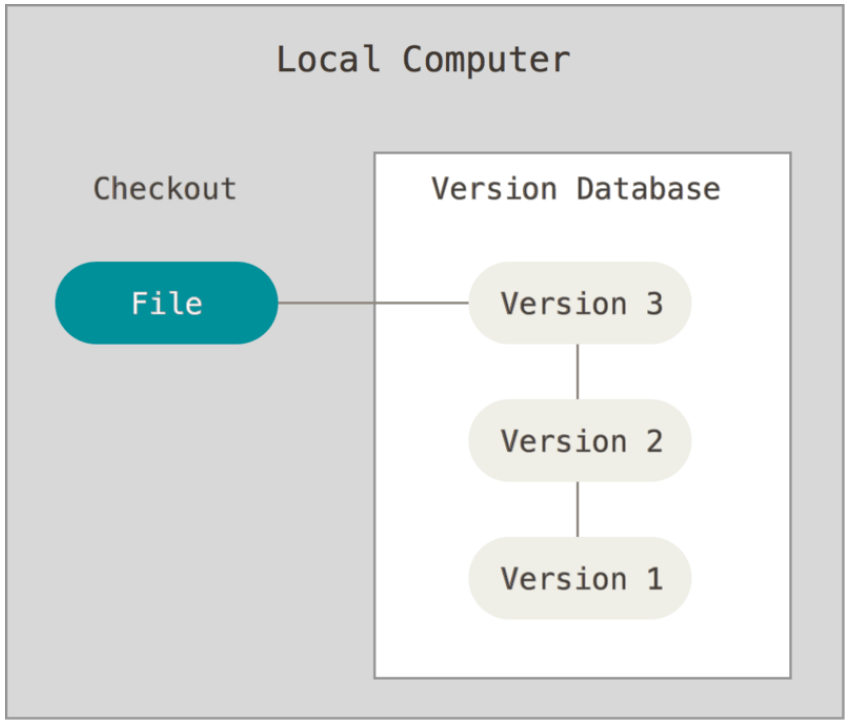
Justification

Keeping track of manually means…

- Significant overhead
  - Organization
  - Documentation
  - Communication
- Error-prone and time-consuming
- Makes collaboration difficult
- Six months from now…???

Version control systems

"Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later (**history**)"



Version control systems

Referring to and tracking back to specific versions

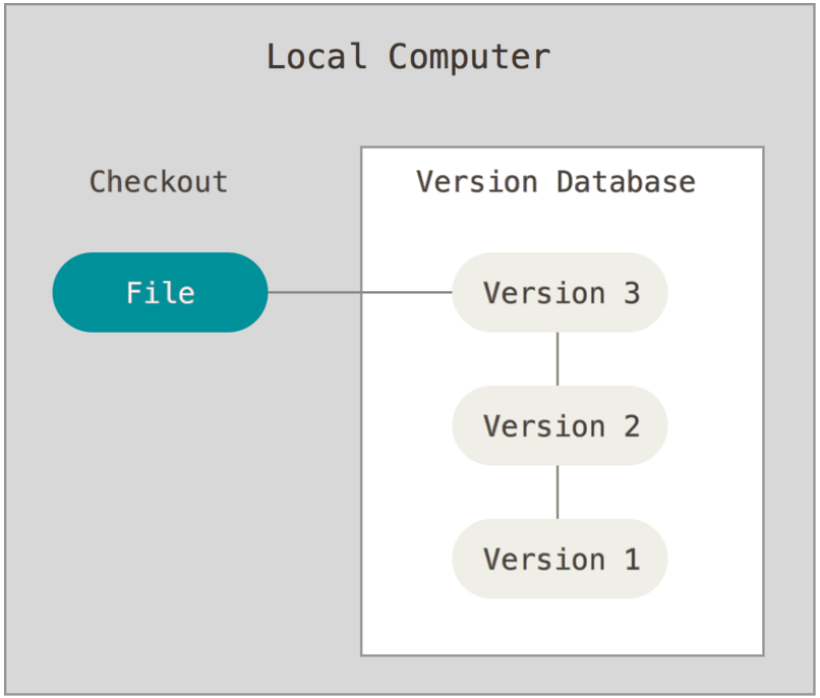
- When was “that” change introduced (backtracking)?
- Includes metadata (who, what, when, why)

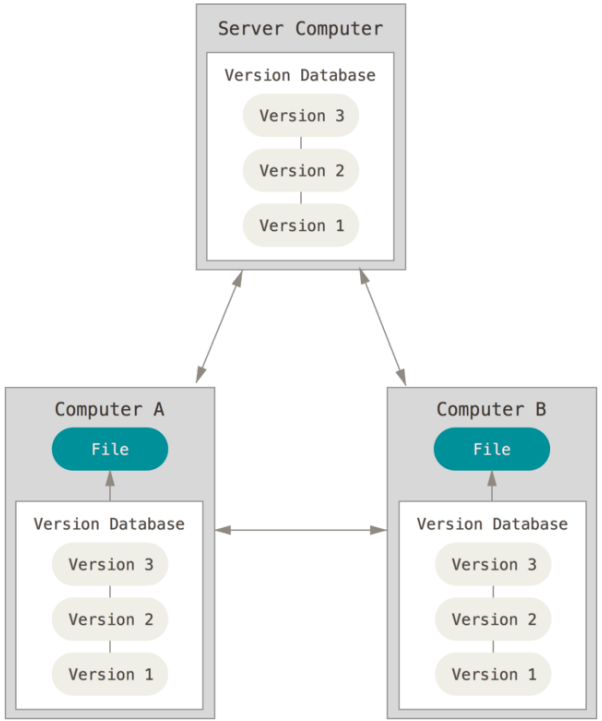
Be organized and maintain long-term control over your codebase

Keep several versions of code at the same time, <u>without clutter</u>

- Keeps known working versions safe
- Experimenting is easy and organized
- Experiments are self-contained
- Experiments can be kept, removed, or merged with the “main” codebase

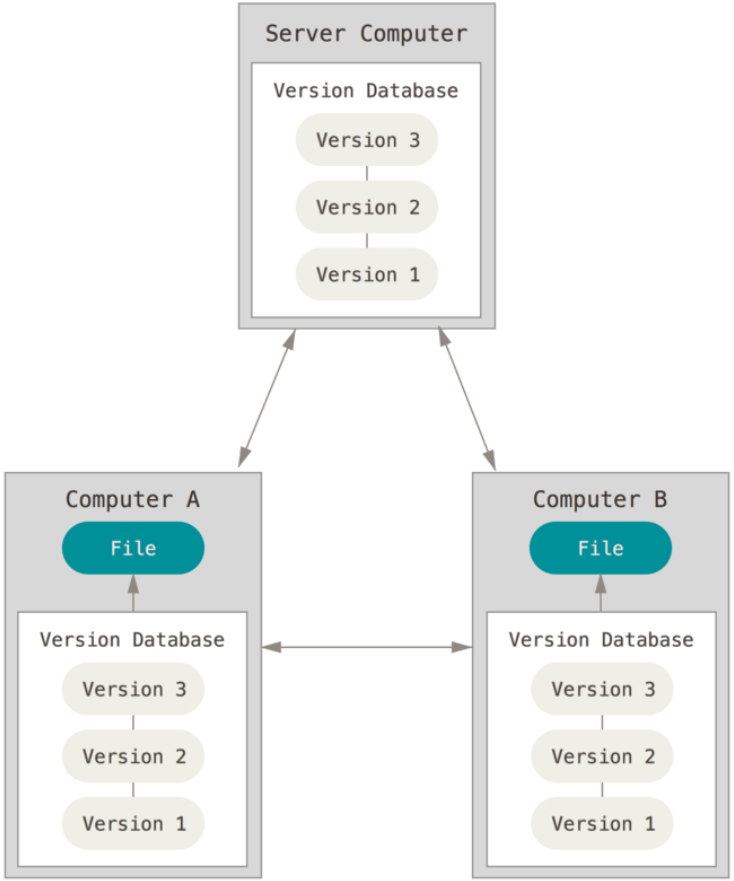
Distributed version control systems





Distributed version control systems

- No single point of failure
- The system keeps everyone synchronized…
- …but asynchronous work is possible
- Group-based model - shared access is default
- Shared responsibility of being up to date



git and GitHub

Git is a DVCS software, created by Linus Torvalds for the management of the Linux kernel.

Other clients exist.

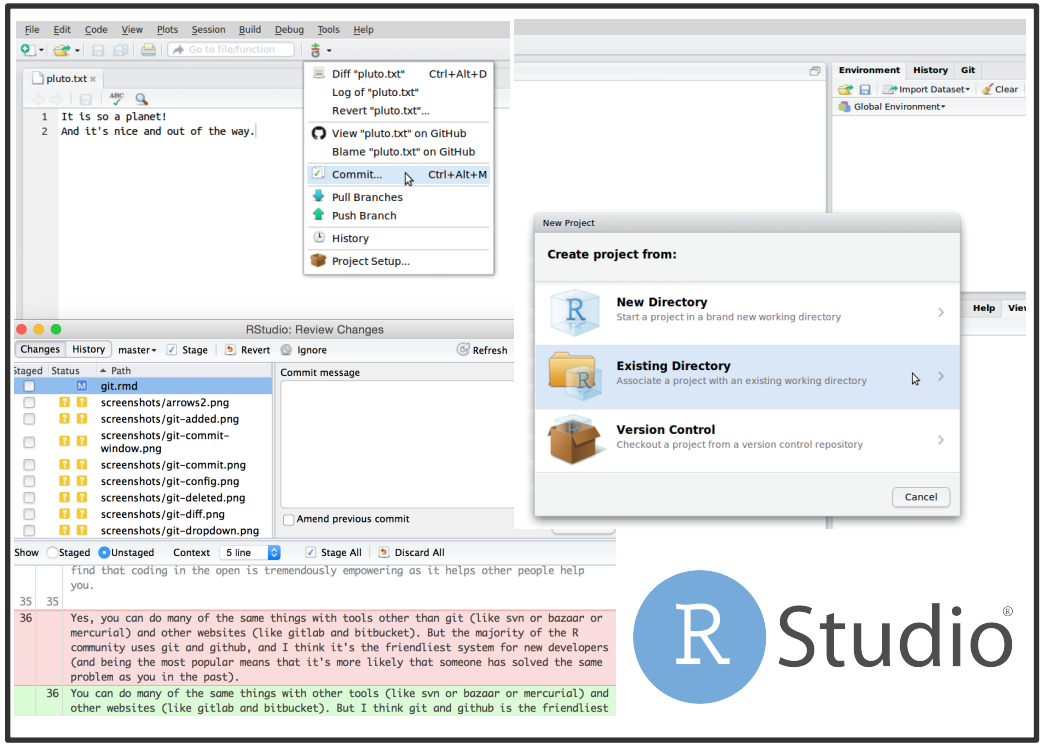
https://git-scm.com

GitHub is a web-based hosting service for version control that uses git

Other services exist.

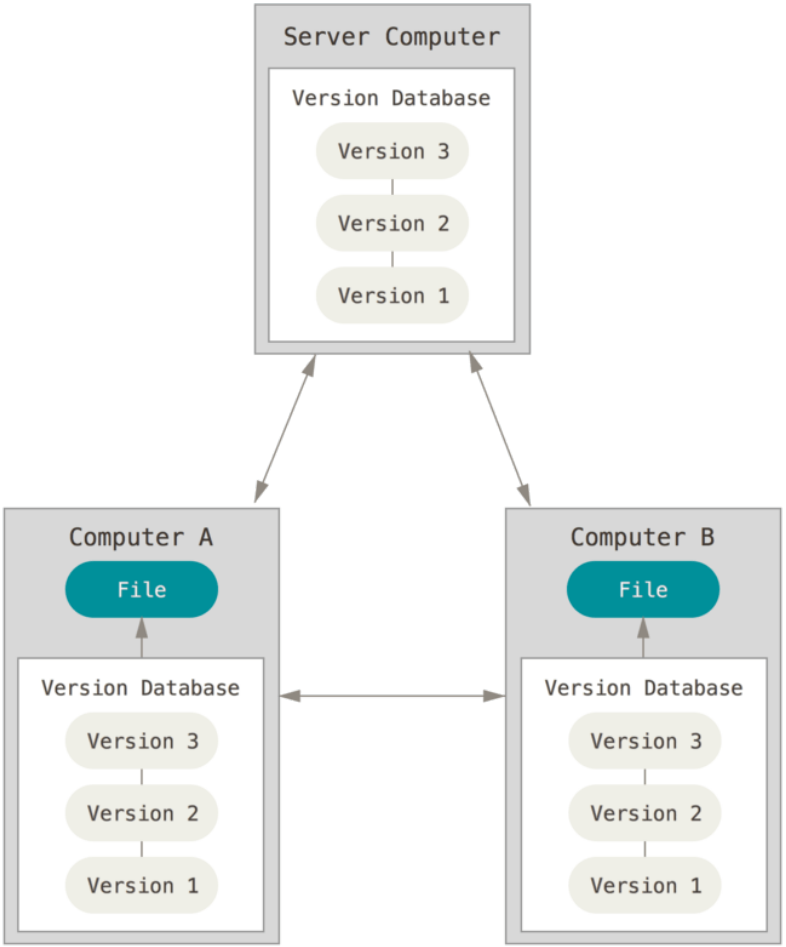
https://github.com

Git with R & Python Editors



- You can work with **git/GitHub** from within code-editors, including Rstudio (R), PyCharm, Visual Studio etc. (Python).
- Changes to scripts can be staged, committed, reverted, and pushed to remote (e.g. GitHub), just as from the command line.

git and GitHub

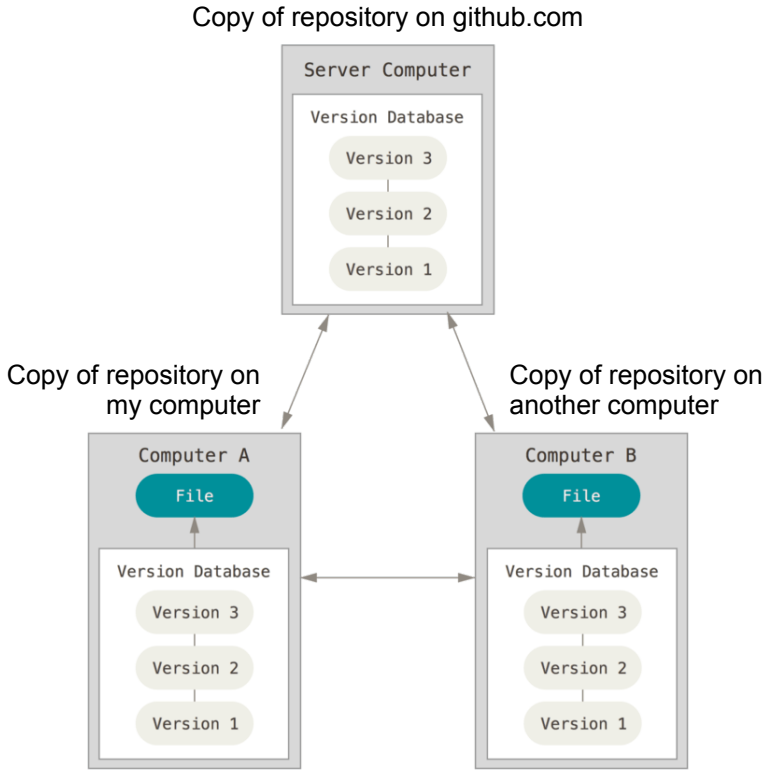


Useful concepts

File tree (working directory):
All the files (organized in directories) that we want to keep track of, originating from a single directory.

Repository (repo)
Place in which changes to the file tree are recorded.
Can be local or remote.
Several copies can exist.

NOTE: We will be working on our local copy first, meaning that there will be no changes on the repos on GitHub, until we decide to synchronize.



Why GitHub?

Helpful web interface:

- Visually browse and explore
- Community aspect (Issues, Wiki, references …)
- Granular access control

Public space: Track records, visibility, transparency, reproducibility

Easy fork/merge system; easy contributing to projects

Remote repository; backup

Shared responsibility of being up to date

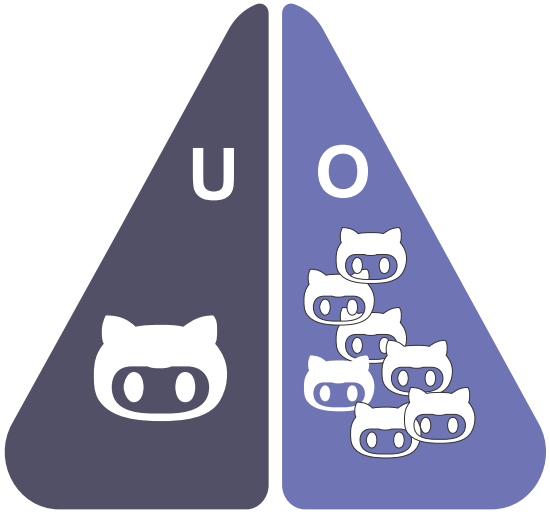
GitHub

GitHub is organized in

- **U** User accounts:

  Personal repositories that have a single owner.

- **O** Organizations:
  - Designed to better accommodate the needs of a team (ex. lab)
  - One or multiple owners and sophisticated access control systems to the repos.



GitHub

GitHub Repositories can be:

PUBLIC

- Accessible to everyone
- Usually released with an open source license
- Still possible to have control over how many people interact with the repo.

PRIVATE

- Not accessible to the public.
- Accessible only from allowed accounts – requires password.

NOTE: Organizations have further options to limit and organize access to repositories

GitHub

MOST POPULAR

Free

The basics for individuals and organizations

- Unlimited public/private repositories
- 2,000 Actions minutes/month
  Free for public repositories
- 500MB of Packages storage
  Free for public repositories
- Community support

$0 per month

Create a free organization

Team

Advanced collaboration for individuals and organizations

- Everything included in Free, plus...
- Protected branches
- Multiple reviewers in pull requests
- Draft pull requests
- Code owners
- Required reviewers
- Pages and Wikis
- 3,000 Actions minutes/month
  Free for public repositories
- 2GB of Packages storage
  Free for public repositories
- Web-based support

$4 per user/month

Continue with Team

Enterprise

Security, compliance, and flexible deployment

- Everything included in Team, plus...
- Automatic security and version updates
- SAML single sign-on
- Advanced auditing
- GitHub Connect
- 50,000 Actions minutes/month
  Free for public repositories
- 50GB of Packages storage
  Free for public repositories

EXCLUSIVE OPTIONS

- Token, secret, and code scanning
- Premium support

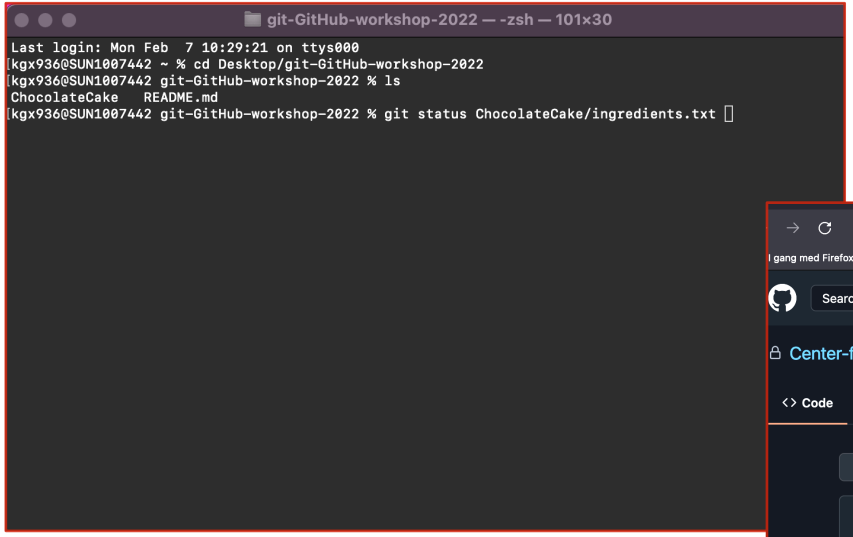
$21 per user/month

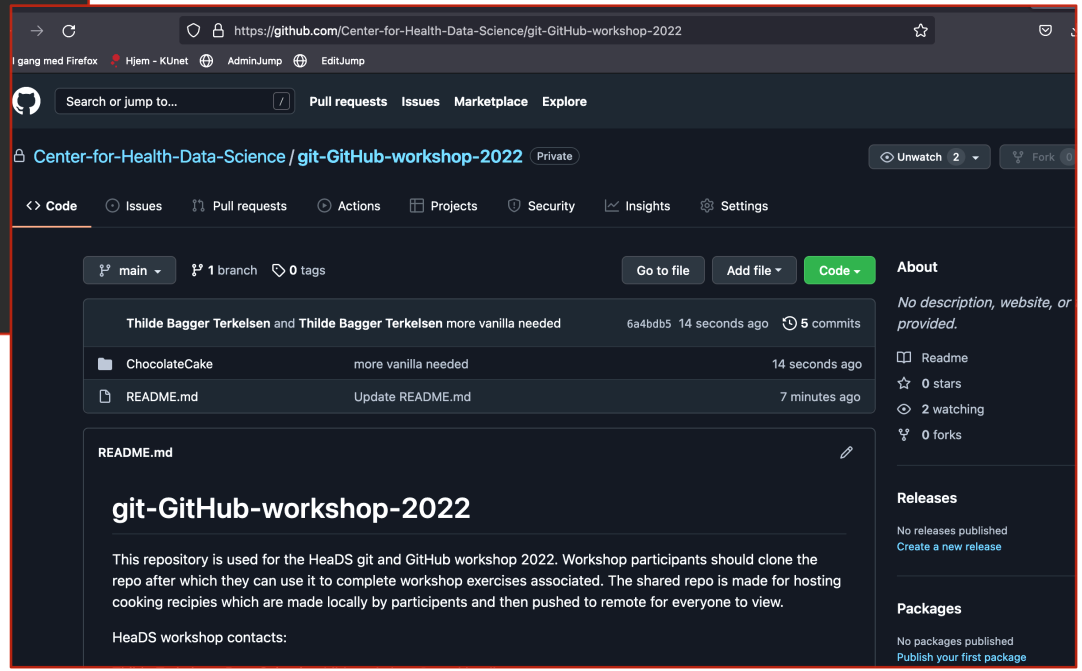
Contact Sales

Start a free trial

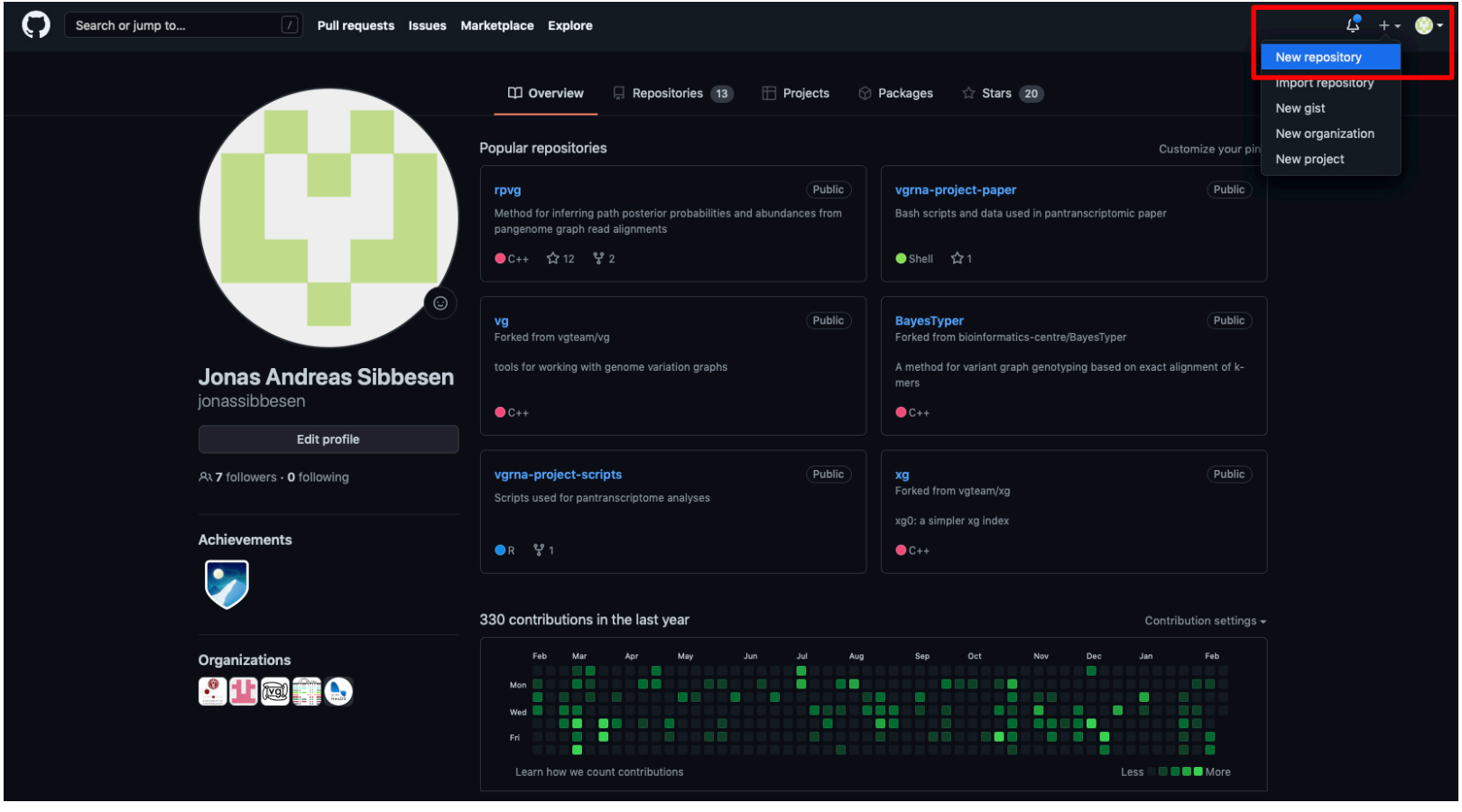
Academics get free “Team” plan!

A look at git and GitHub

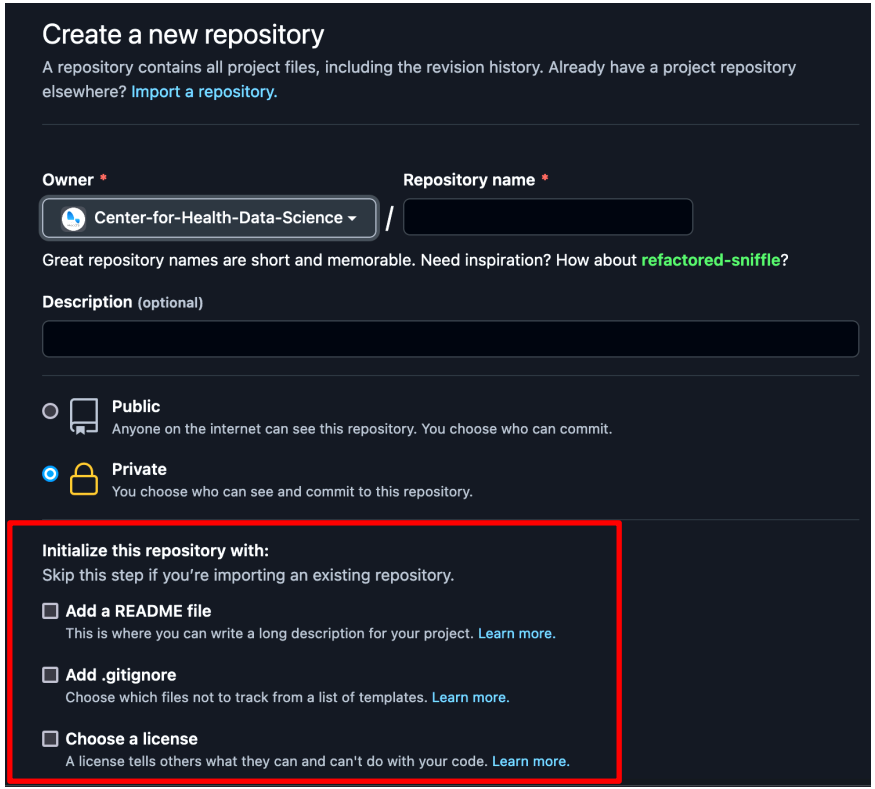




Let’s create a GitHub repo



Let’s create a GitHub repo



README file
File shown by default when someone visits the repository – good for an introduction to what he repository is about. Git markdown format.

.gitignore
Hidden file containing specifications of files/types that should be ignored at all times by git. There are some default choices.

LICENSE
Creates a LICENSE file that contains the license of the content of your repository.

https://choosealicense.com

Choose an open source license

An open source license protects contributors and users. Businesses and savvy developers won't touch a project without this protection.

Which of the following best describes your situation?

I need to work in a community.

Use the **license preferred by the community** you're contributing to or depending on. Your project will fit right in.

If you have a dependency that doesn't have a license, ask its maintainers to **add a license**.

I want it simple and permissive.

The **MIT License** is short and to the point. It lets people do almost anything they want with your project, like making and distributing closed source versions.

Babel, **.NET Core**, and **Rails** use the MIT License.

I care about sharing improvements.

The **GNU GPLv3** also lets people do almost anything they want with your project, *except* distributing closed source versions.

Ansible, **Bash**, and **GIMP** use the GNU GPLv3.

Personal Access Token (PAT)

Personal access token to use with the command line.

Token is used to “communicate” between local repo and remote GitHub repo.

Advantage of PAT over password:

- **Unique**: Can be generated per use or per device.
- **Revokable**: Can revoke access to each one at any time.
- **Secure**: Random strings of characters that cannot be attacked by brute force i.e. less hackable than a password.
- **Quantity**: Any number of access tokens can be created, compared to only one password per user.

git Syntax

We will use the terminal command `git`. Syntax:

```
git <command> [<options>] [<args>]
```

Args are usually names or files or directories.

Individual commands have help pages:

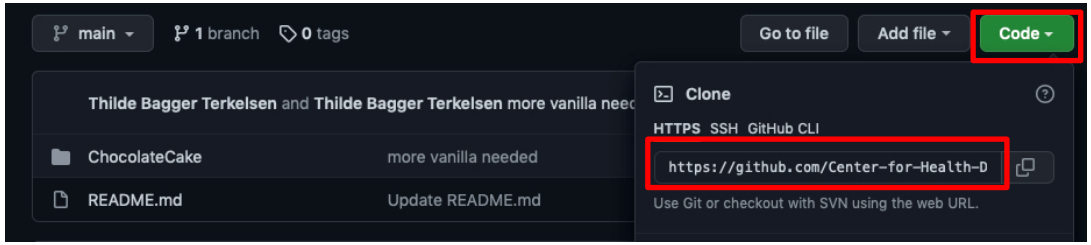
```
git <command> --help
```

Examples:

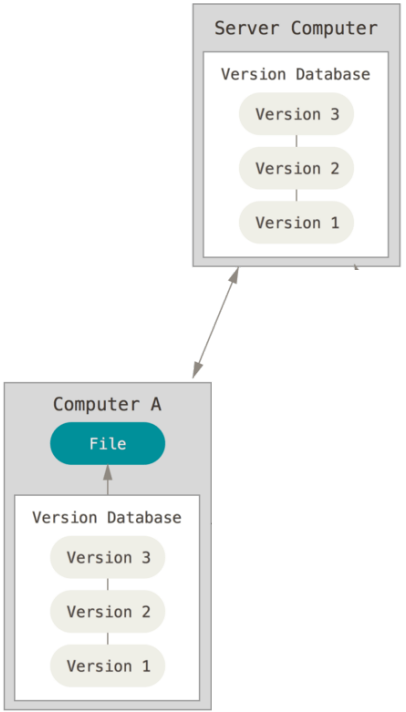
```
git add file.txt
git commit -m “changes were done”
git branch -d name
```

Cloning a GitHub repository

We need a local copy of our repository :



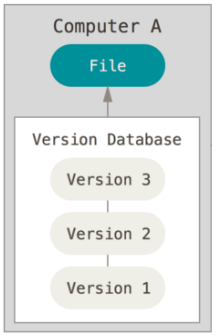

```
git clone <GitHub link>
Username: <your user>
Password/Token: <your PAT>
cd <your repo>
```



Creating a git repository

You could create it from scratch as well:

```
mkdir my-repo; cd my-repo
git init
```

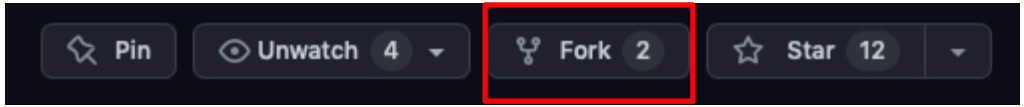


N.B: that in this case we would not have a remote! Creating a repository on GitHub and cloning it ensures that:

- We have a remote repository
- Remote and local copy are linked

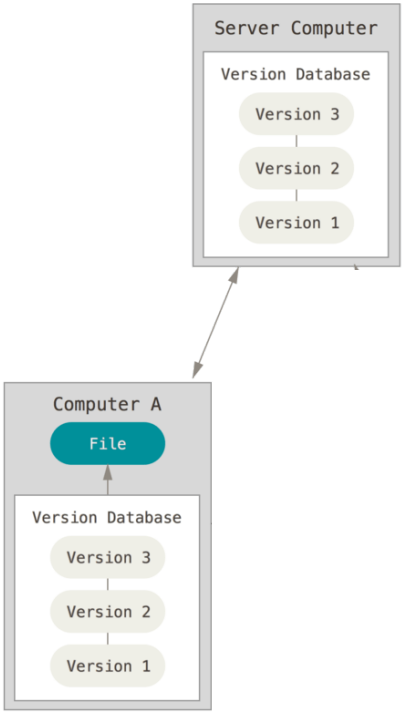
Forking a GitHub repository

Create a you own copy of someone else's repository



Forks are most commonly used to:

- Propose changes to another persons repository
- Use another persons repository as a starting point for own project



Checking status of a git repo

- Check the current state of the repository:

```
git status
```

- See the name and location of the remote repository:

```
git remote -v
```

Pre-Exercise

Exercise
0

Go to your GitHub account and set up your personal access token. Follow the small step-by-step below:

https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token

Important: Copy your access token and note your GitHub username as you will need both for Exercise 2.

Exercises 1-3

Exercise 1

Create a new public repository on GitHub to hold your favorite cooking recipes.

Exercise 2

Create a local copy of your GitHub repository by **cloning** it in a folder on your computer. Make note of the location of the directory that you choose to host your local copy.

Exercise 3

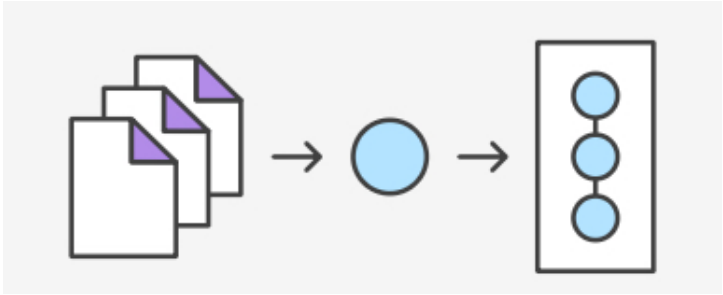
Move to local repository. Check the status of this repository (don't worry if all the information does not make sense yet). Check name and location of the remote repository.

Useful concepts

Commit:
Snapshot of a particular state of a file tree.

Stage:
Selection of changes made on a repository to be added to a commit.

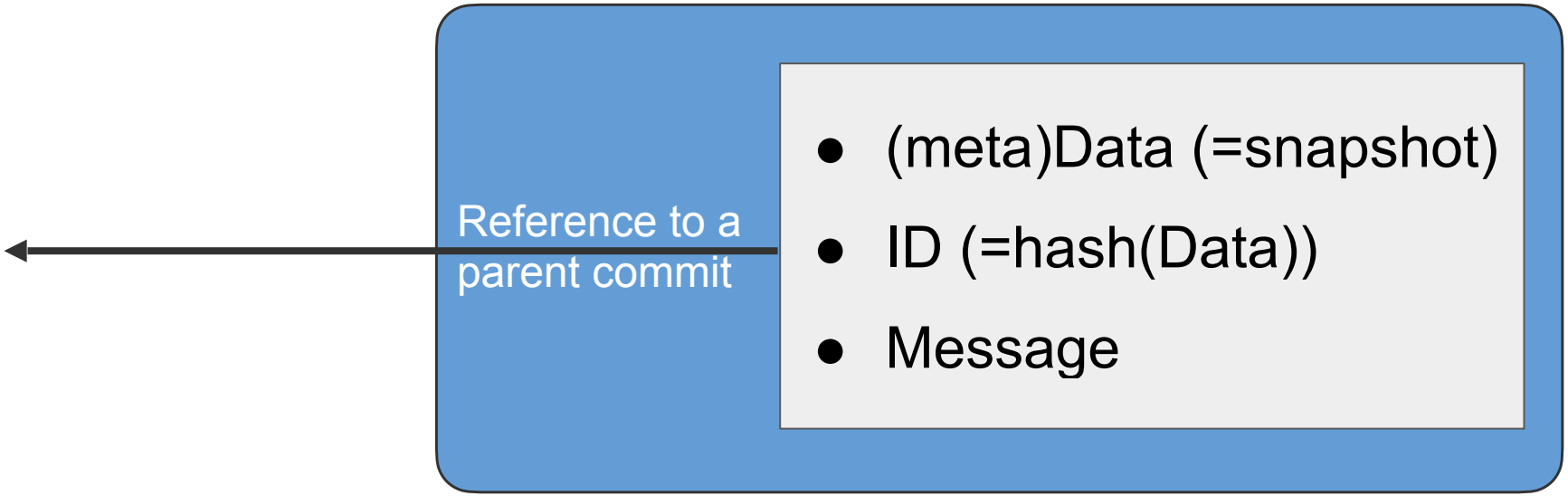
head & branch:
Reference (“pointer”) to a particular commit.



https://www.atlassian.com/git/tutorials/saving-changes

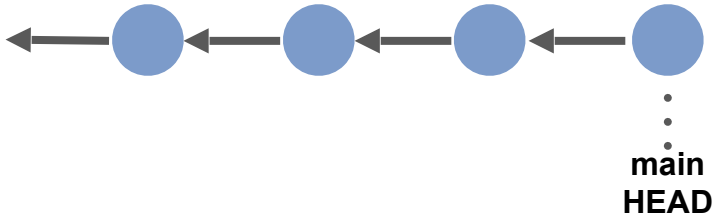
Useful concepts

Anatomy of a commit



Useful concepts

Anatomy of a commit



Be aware of the “**HEAD**” head!
It is the head referring to the commit currently checked out in the file tree

origin = original remote repository

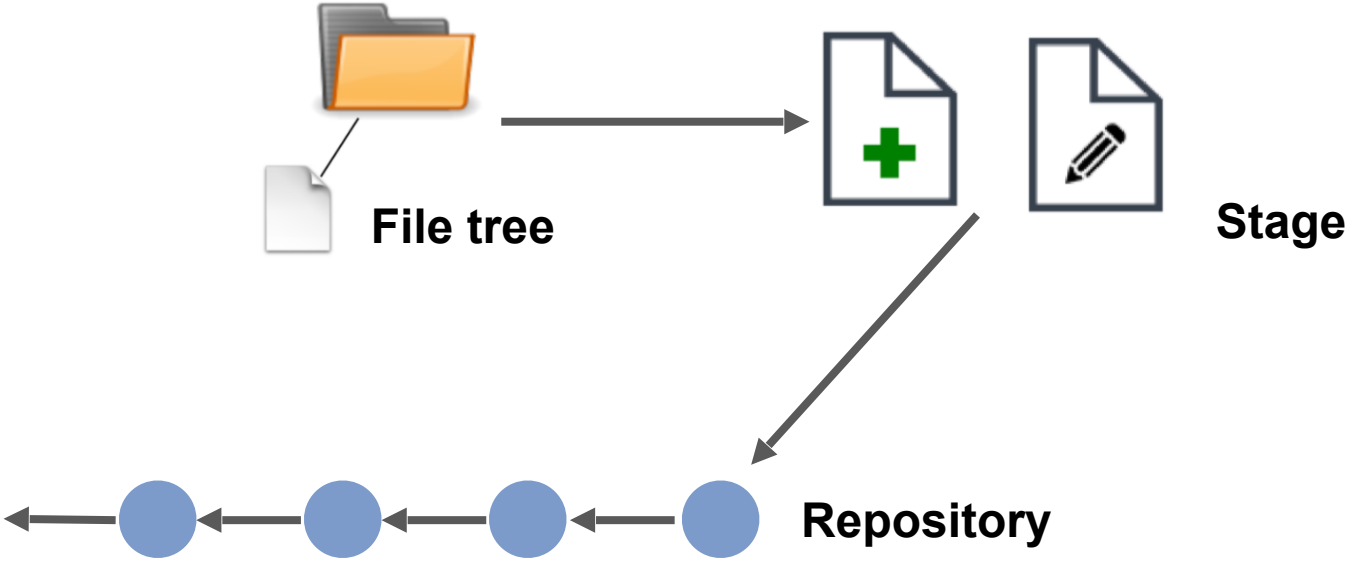
main = main branch (local and remote)

HEAD = current commit you are viewing

master = previous name for remote main (changed in 2020)

Useful concepts

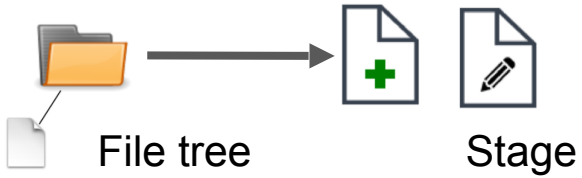
Workflow of a commit



Adding files to stage

Files are added to stage using the command **`add`**

```
git add <filename>
```



- Allows to stage files or even directories (if they contain files!)
- Option `-A` stages all changes if no arguments are given
- More than one filename are allowed including wildcards

```
git rm <filename>
git mv <filename>
```

First Commit

When you're happy with your changes and have add all the files to stage the next step is to commit the changes:

```
git commit

git commit -m "Commit Message"
```

- Use `-a` to automatically stage files that have been modified and deleted. **New files you have not told Git about are not affected.**

```
git commit -a -m "Commit Message"
```

Good Practices

- **Commits** should be:
  - Atomic and self-contained (or WIP), but not excessively pedantic
  - Done often (try not to have gigantic blob commits)

- **Commit messages** should:
  - Always have a short descriptive comment (no “bugfix”, “save”, etc.)
  - Written to be understandable by someone else >1 year down the line, including you :)

Checking history

To check the history of a repository:

```
git log
```

A few useful options:

- To see only the commits of a certain author

```
git log --author=”SomeAuthor”
```

- To see a very compressed log where each commit is one line

```
git log --oneline
```

- To see a more complete history in a graph form:

```
git log --all --decorate --graph --oneline
```

Checking differences

Check

- not staged changes since the last commit:

```
git diff
```

- staged changes since the last commit:

```
git diff —cached
```

- differences between two commits:

```
git diff <ID1> <ID2>
```

Exercises 4

Exercise 4

- Add a recipe to your repository by creating an *ingredients.txt* and a *recipes.txt* file (can contain anything you would like) in the repository folder.
- Check the status of the repository. Stage the new files using **add** and check the status. **Commit** the staged files and check the status.
- Create a directory with the recipe name and move your files to it; create a commit with your changes.
- Add or remove one step in *recipes.txt*; create a commit with your changes.
- Keep experimenting with staging and committing.

Exercises 5

Exercise
5

- Check the commit history. Can you identify the author, ID, time and date of your commits?
- Try experimenting with the different ways of showing the history.
- Check the difference between different commits. Can you identify the filename(s) and changes?

Back to the past

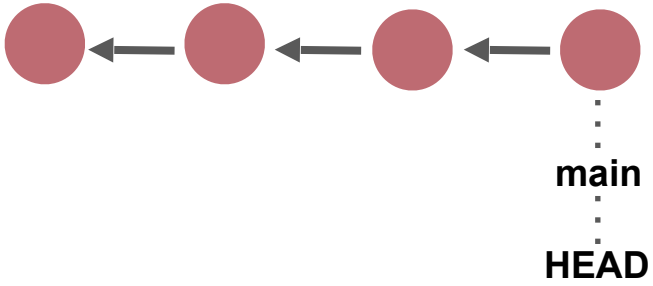
We can move around the repository using the command **`checkout`**

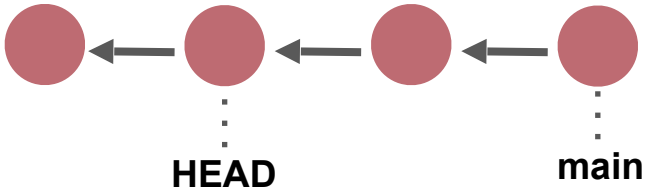
```
git checkout <identifier>
```

Identifier can be:

- A commit **ID**
- A branch **Name**

Changes the local content of the file tree to that of the reference - which means, moves **`HEAD`** to identifier.





Back to the past

Rewriting history is (usually) a bad idea

Every change you do in git is done with a forward in time perspective - the past should be changed only in some specific occasions!

- Changes (even to something done a long time ago) need to be done in commits added to the HEAD
- There are other mechanisms to re-start from an earlier commit (i.e. branching/merging)

I’ve botched staging or commit!

How to fix the last commit message:

```
git commit --amend -m “new message”
```

I’ve botched staging or commit!

`reset` is useful to “undo” some of the changes:

`git reset`

- Removes all staged changes.

`git reset <filename>`

- Removes file from staged area.

`git reset --hard <ID>`

- Moves reference to that commit (can be HEAD) and resets the working directory to match that commit.

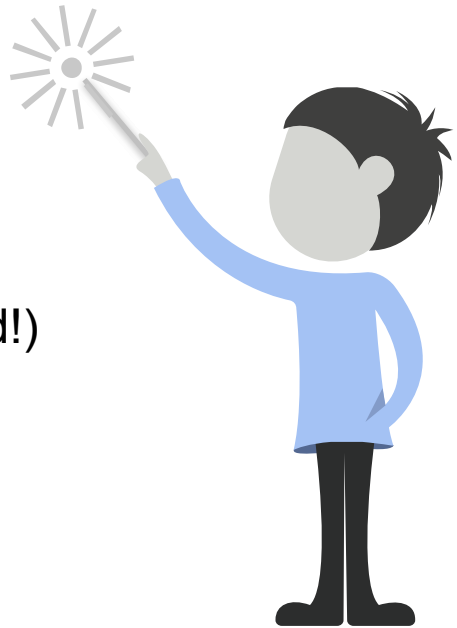
Note that these are potentially **destructive operations! Only on local branches**!

I’ve botched staging or commit!

`revert` is used to add commits that “reverse” the changes done by older commits:

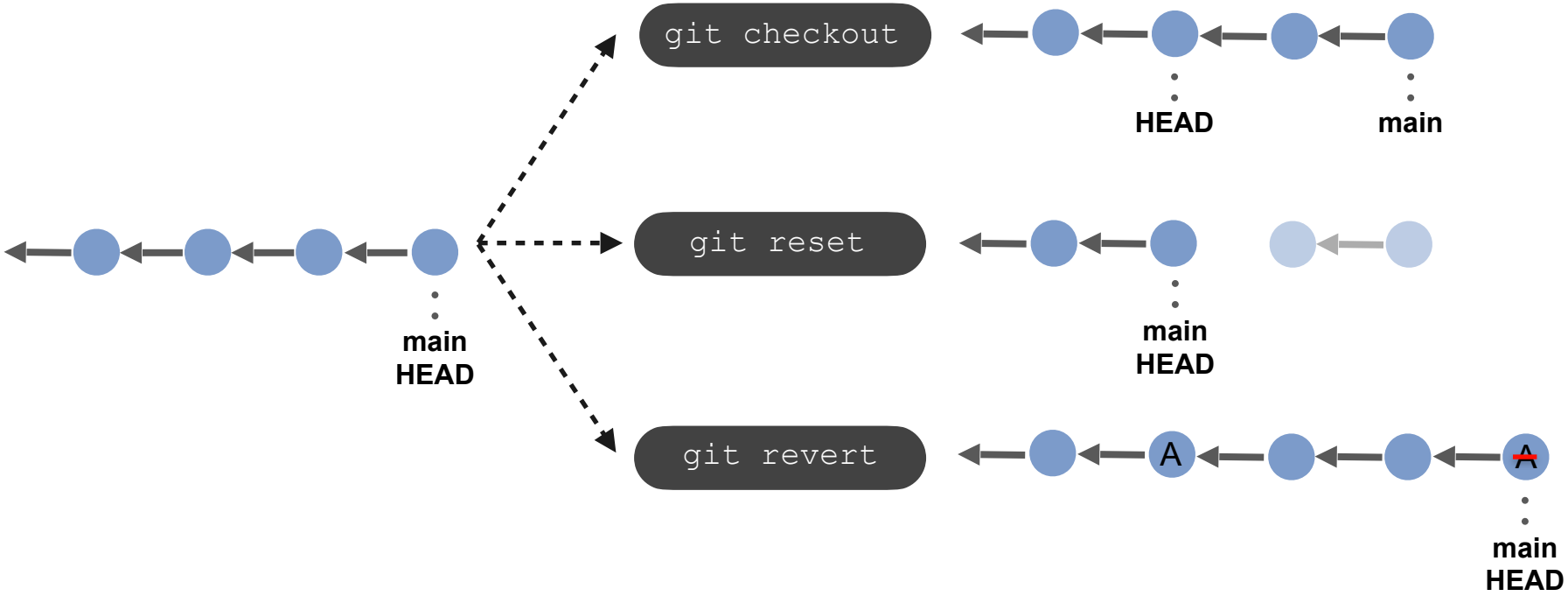
```
git revert <ID>
```

```
git revert <ID1>..<ID2>
```



- Non-destructive operation (history is maintained!)
- Here “ID” is the commit that you want to revert.
- History is polluted with commits that just revert – not the cleanest option
- **Better than reset for shared repositories**

Git checkout, reset & revert



Exercise 6

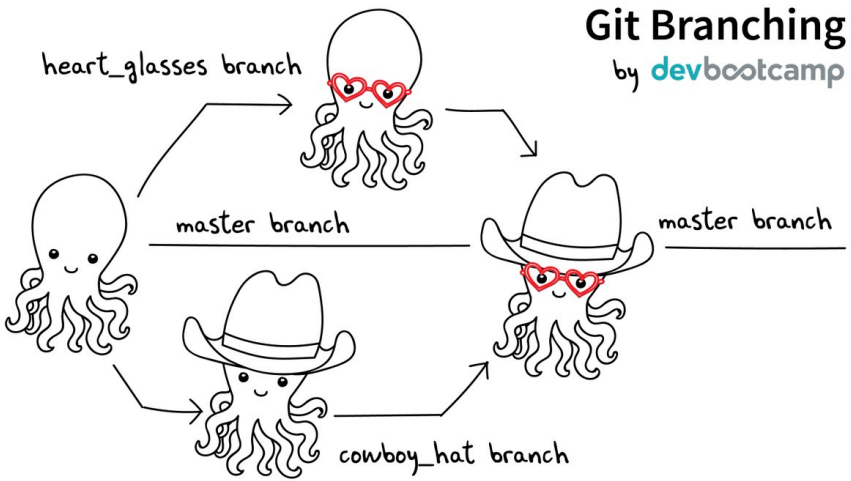
Exercise
6

- Change the last commit message.
- Checkout an earlier commit. Check the content of the files. Check the history and status of the repository (do not commit). What do you notice?
- Checkout the last commit (use **main** as identifier).
- Add a step in *recipes.txt* and add the changes to stage, but **do not commit**. Check the status. Do a hard reset of **HEAD** and check the files and status. What do you notice?
- Add a new ingredient to *ingredients.txt* and commit the changes. Add a new step to *recipes.txt* and commit the changes.
- **Revert** the first of the two new commits (it will ask you to write a commit message). Check the files and history. What do you notice?

Branching

A **branch** represents an independent line/track of work on your repository.

- Keep a branch that is in working order (main branch).
- Isolate experiments from the main branch.
- Work on more than one feature at the same time, independently.
- Merge changes we like into the “main” branch.



Branching

- List of all the branches in your repo

  `git branch` synonymous to

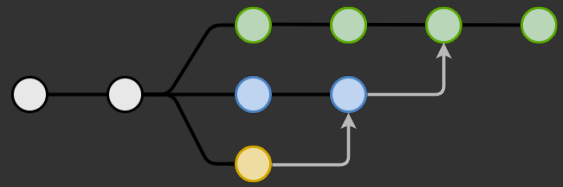
  `git branch --list`
- Create a branch named <branch>

  `git branch <branch>`
- Switch to a branch

  `git checkout <branch>`
- Safe deletion: prevents from deleting the branch if it has unmerged changes

  `git branch -d <branch>`
- Force delete

  `git branch -D <branch>`



Good Practices

- Branches should:
  - Have descriptive names.
  - Done every time you want to experiment or add a new feature, bugfix etc.
  - Have a single purpose (one feature, one bug, one change).
  - Derived from a sensible commit (often the latest commit in the main branch)

Merging

Two main cases for merging

We want to integrate a feature we were working on in a branch, into main.

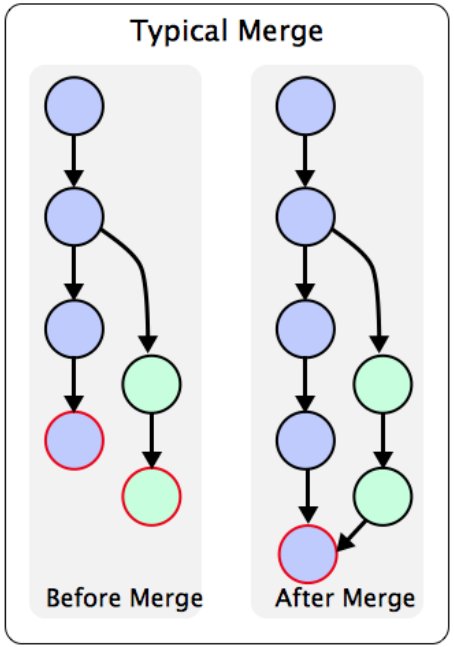
Main has diverged enough from our branch that we want to integrate the new commits in it before merging our feature – so we merge main into our feature branch.

Merging

Merging incorporates changes made in another commit/branch **into the current/active branch.**

- **Switch** to the branch you want to change (not the one you want to merge!)
- **Merge** incorporates the changes made by the other branch **into the current/active branch.**

```
git checkout <active branch>
git merge <branch to merge>
```



Exercise 7

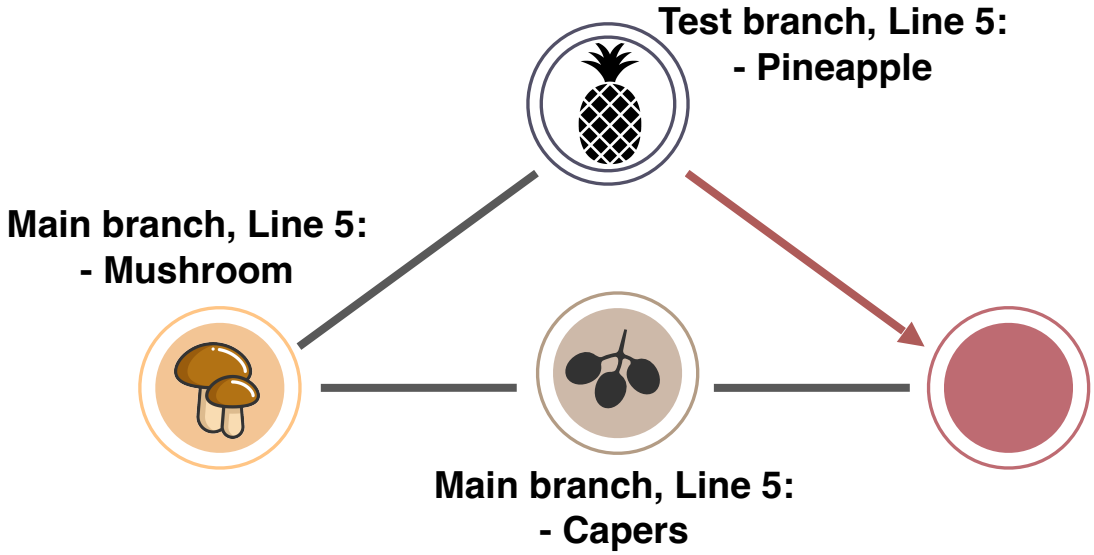
Exercise
7

- Checkout the last commit of your **main** branch if needed.
- Create an **experimental** branch and move to it. Change an ingredient in *ingredients.txt* and store the change as a commit.
- Go back to your **main** branch and add a step in *recipes.txt* and store the change as a commit.
- Merge your **experimental** branch into your **main** branch (it will ask you to write a commit message). Check the history (try using the graph mode).

Conflicts

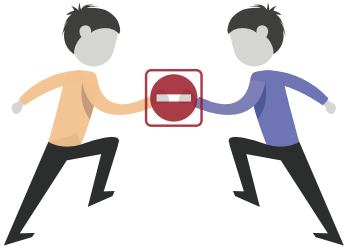
A merge **conflict** happens when trying to merge branches that have changes **in the same files and in the same lines**

Merge test into main: which version should we keep?



Conflicts

```
(base) Matteos-MacBook-Pro-2:test matteotiberti$ git merge test
Auto-merging recipe.txt
CONFLICT (content): Merge conflict in recipe.txt
Automatic merge failed; fix conflicts and then commit the result.
```



Main branch **Test branch**

- Check which file(s) have conflicts
- Modify files by hand to achieve the desired result.
- Add all files once they are ready and commit. This solves the merge in a commit.
- **Conflict markers**! Remember to remove all markers and change your code in the final version you want.

```
- tomato
- mozzarella
- flour
- water
<<<<<<< HEAD
- capers
=======
- pineapple
>>>>>>> test
```

Exercise 8

Exercise
8

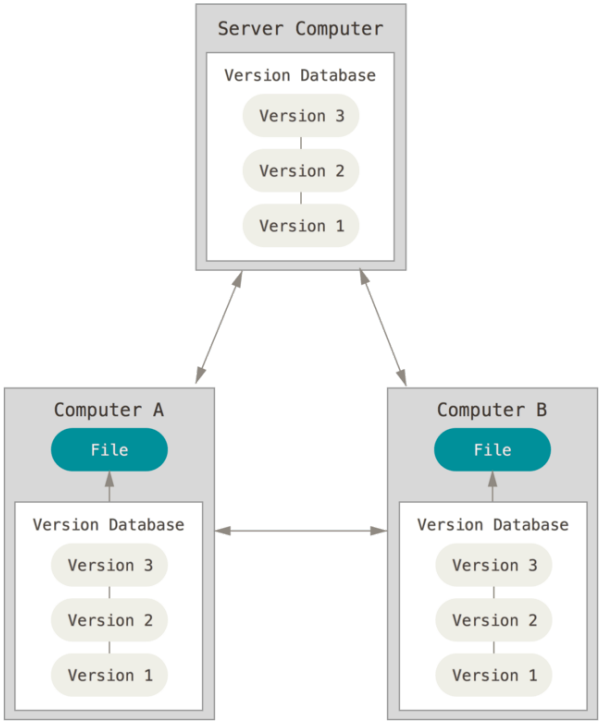
- Switch to your **experimental** branch and change the first line in one of your files, and commit.
- Switch to your **main** branch and change the first line of the same file in a different way, and commit.
- Try to merge your **experimental** branch into your **main** branch; you should get a conflict. Check the status of the repository.
- Edit the conflicting file and solve the conflict. Stage and commit to fix it.

Interacting with a remote

A **remote** is a repository that is linked to our local (or in our case, our local repository started out as a copy of a remote repository).

We can create a local copy of our repository, make our changes as commit, and submit those to remote

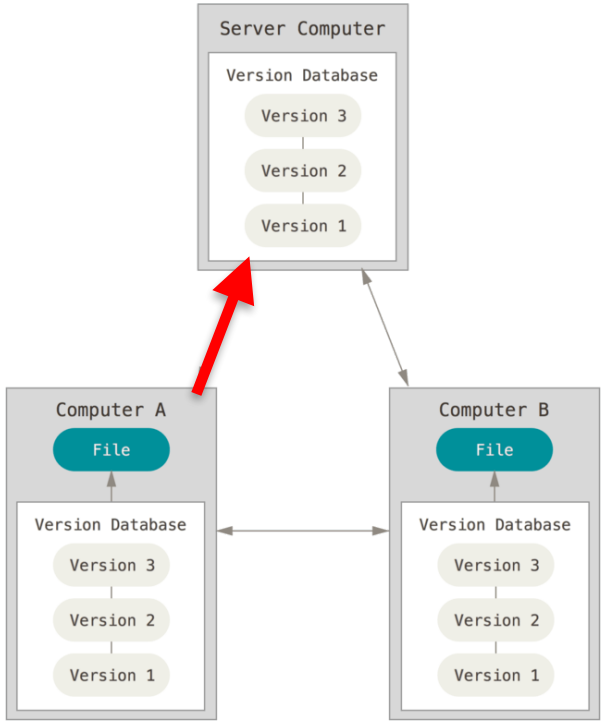
We should take into account the changes that happened in the meanwhile on remote (other people's)



Interacting with a remote

- Use **`push`** to upload local repository content to remote (i.e. export commits to remote branches)

```
git push <remote> <branch>
```



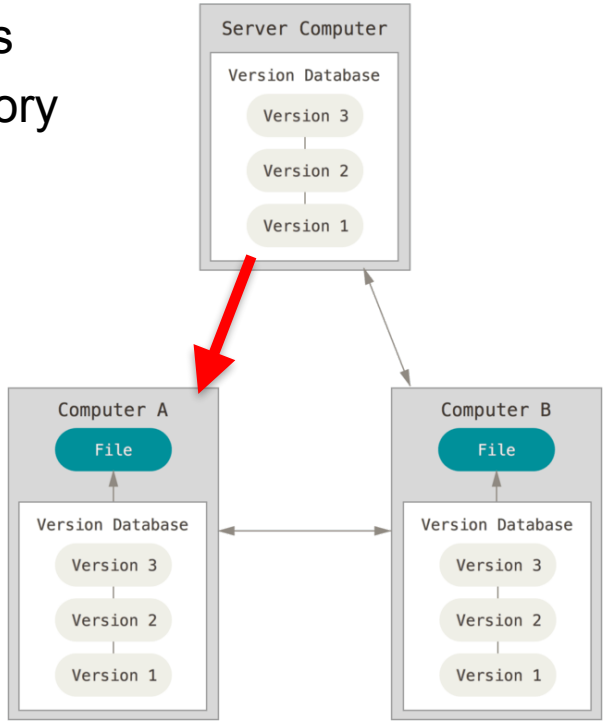
Interacting with a remote

- Use **`pull`** to download and apply the changes that were done in remote to your local repository (i.e. apply commits by other people):

```
git pull <remote> <branch>
```

- Use **`fetch`** to download without merging (remote fetched branches are named as "`<remote>/<branch>`"):

```
git fetch <remote>
```



Exercise 9

Interacting with a remote

Exercise 9

- Push your changes to the remote repository for your **main** branch. Visit your remote repository on github.com. Check if your changes are there.
- Create a new branch. Add a couple of commits to the new branch. Push this new branch to your repository. Check the changes on GitHub. What do you notice?
- Merge your new branch into **main**. Push your changes and check them on GitHub. What do you notice?

Exercise 9 cont.

Interacting with a remote

Exercise

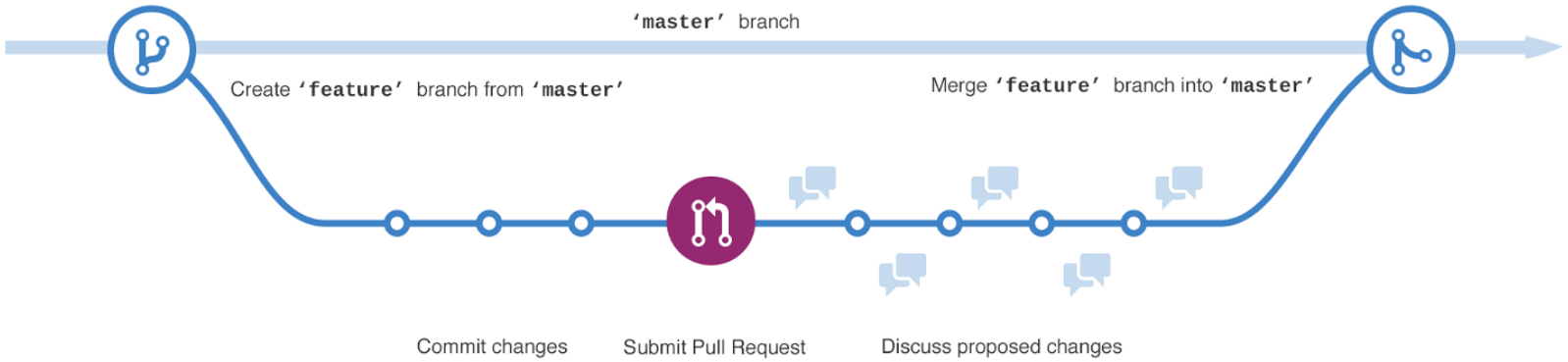
9

- Edit the **README file on GitHub**. Create a new commit on **main** and try to push it. What happens?
- **Fetch** the changes from the remote repository. Note that the remote version of the **main** branch is saved as **origin/main**. Look at the difference between the local and remote version of **main**.
- Merge remote version **origin/main** into the local version of **main** and push the changes to the remote repository on GitHub. Visit GitHub and check if your changes are there.

A deeper look at GitHub

Contributing and interacting with other users: Issues and Pull requests

- Projects on GitHub often follow a pattern of development and communication.
- The main branch is considered the “stable” branch and is not changeable directly - only through merges and pull requests.
- Contributions happen in branches that are merged into the main.
- Releases happens once in a while from the main branch.



A deeper look at GitHub

Interacting with other users: Issues and Pull requests

Read the documentation of the project you want to contribute to.

README.md

git-GitHub-workshop-2022

This repository is used for the HeaDS git and GitHub workshop 2022. Workshop participants should clone the repo after which they can use it to complete workshop exercises associated. The shared repo is made for hosting cooking recipies which are made locally by participents and then pushed to remote for everyone to view.

HeaDS workshop contacts:

Thilde Terkelsen, Data Scientist thilde.terkelsen@sund.ku.dk

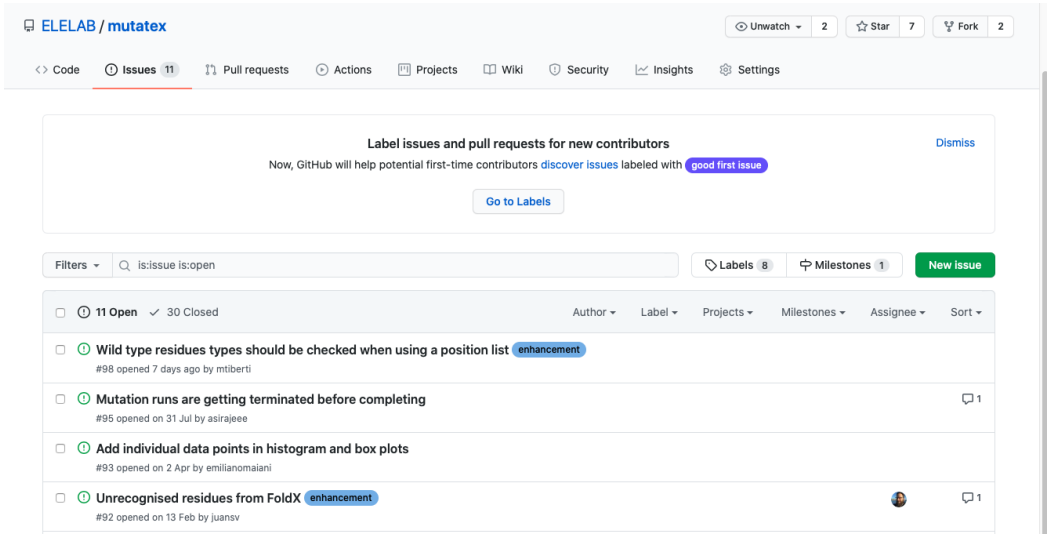
Jonas Sibbesen, Assistant Professor josi@di.ku.dk

Diana Andrejev, Data Scientist andrejeva@sund.ku.dk

A deeper look at GitHub

Interacting with other users: Issues and Pull requests

Check *Issues* if you have a problem or question to see if others have already asked the same. If needed, open a new issue.



A deeper look at GitHub

Interacting with other users: Issues and Pull requests

Create a local copy of the repository:

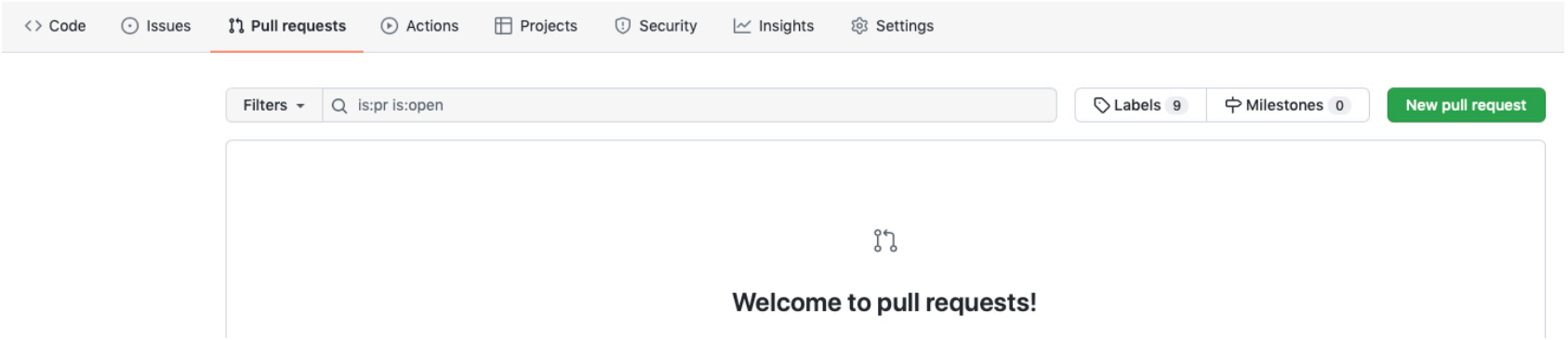
- **Clone** if you have write access to the repository (direct access as *collaborator* or part of the repo *organization*).
- **Fork** if you have no affiliation to the repository.

Create a **new branch** from the main branch. Add changes as commits. Push them so they are present on the remote (GitHub).

A deeper look at GitHub

Interacting with other users: Issues and Pull requests

Once ready, create a *Pull Request (PR)*. Your code might be subject to code review. If so, the maintainers will decide if/when your code is good enough to be merged.

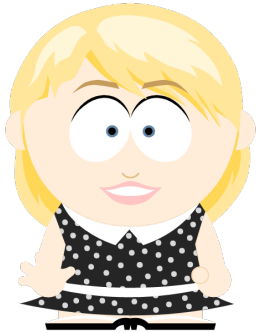


Exercise 10

A deeper look at GitHub

Exercise 10

- Go to the web page of the shared-recipes repository for which you have become a collaborator.
- Open an issue about adding your own recipe.
- **Clone** your local copy of this shared repository. Create a new branch. Add your recipe. Commit and push your changes. Try to pull main and merge if needed before pushing.
- Go back to the repository webpage and open a **Pull request** with your new branch. You might need to re-merge main because of the changes.
- Wait for one of us to approve the pull request. Pull from the repo and check our recipe book.



Thank you for today

See you at HeaDS