



INTRO - DUCTION TO GIT AND GITHUB

DATA SCIENCE LAB
AHMAD ALHINEIDI



git






What to Expect?

- The goal of using a version control system
- Basic Understanding of Git's internal model
- Basic Git commands
- Use Git to collaborate with others
- Get and publish changes from remotes (like a repository on GitHub)
- Become familiar with using command line interface
- Become familiar with Github
- Be able to publish website using Github pages





Agenda – day 1

- 01:15 – 02:00: Introduction
- 02:00 – 03:00: Version Control System (Types – git – how it works)
- 03:00 – 03:25: Break
- 03:25 – 04:00: Basics of Linux
- 04:00 – 05:00: Getting started with GIT





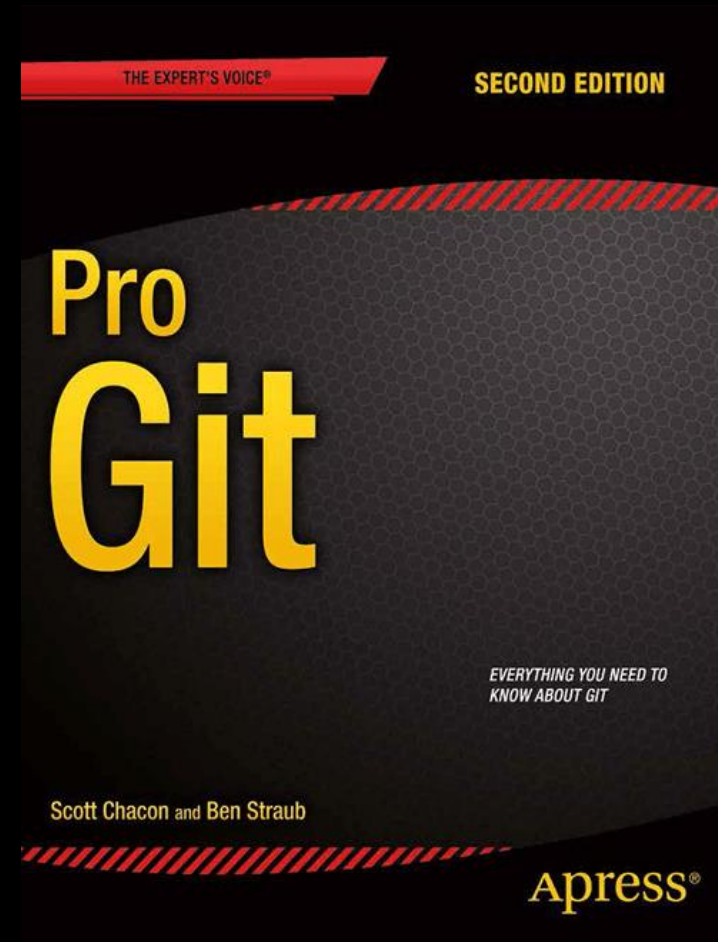
Agenda – day 2


- 01:15 – 02:30: Git core functionalities
- 02:30 – 03:00: Working with remote (github, maybe github pages?)
- 03:00 – 03:25: Break
- 03:25 – 05:00: team work and wrapup





READING RECOMM- ENDATION





Version Control System (VCS)

- Record changes to directory, files, code
- Essential tool when collaborating with others
- Good-practice to use even without working with others
- Git is the most used VCS
- Created by Linus Torvaldes
- Git is a free and open-source software
- It is simply a file inside your directory



Other VCS

- Apache Subversion (svn)
- Mercurial
- Perforce Helix Core
- Monotone
- Others





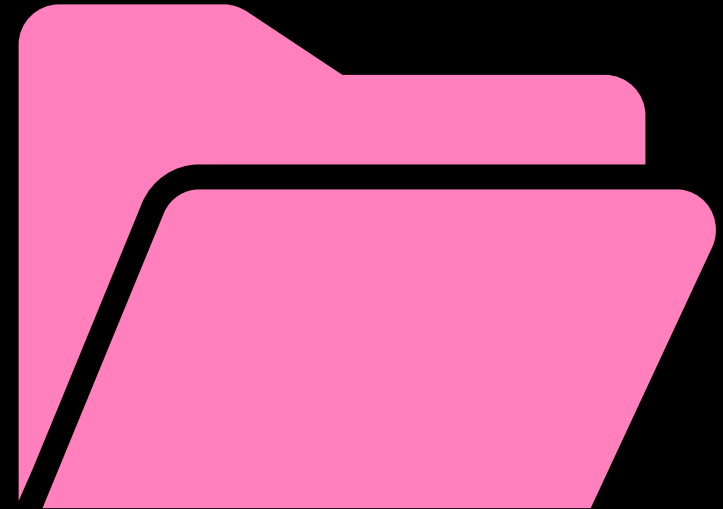
Discussion

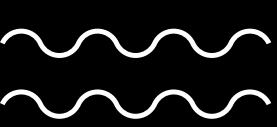
- What are the pros and cons of using VCS? (Talk to your neighbor. They might have some wisdom!)



VCS Approaches

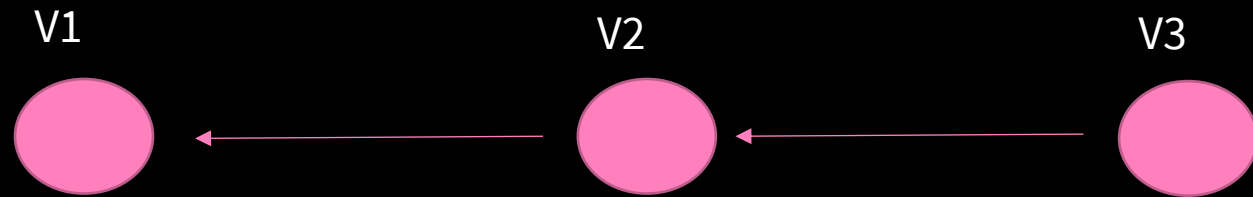
- Basic: make copies of folder with time stamps
- Manually edit changes from others into your code
- Ends up with final copy or version to use
- Project_final
- Project_really_final
- Project_really_final_last
-
- Project_really_final_last_clean_5





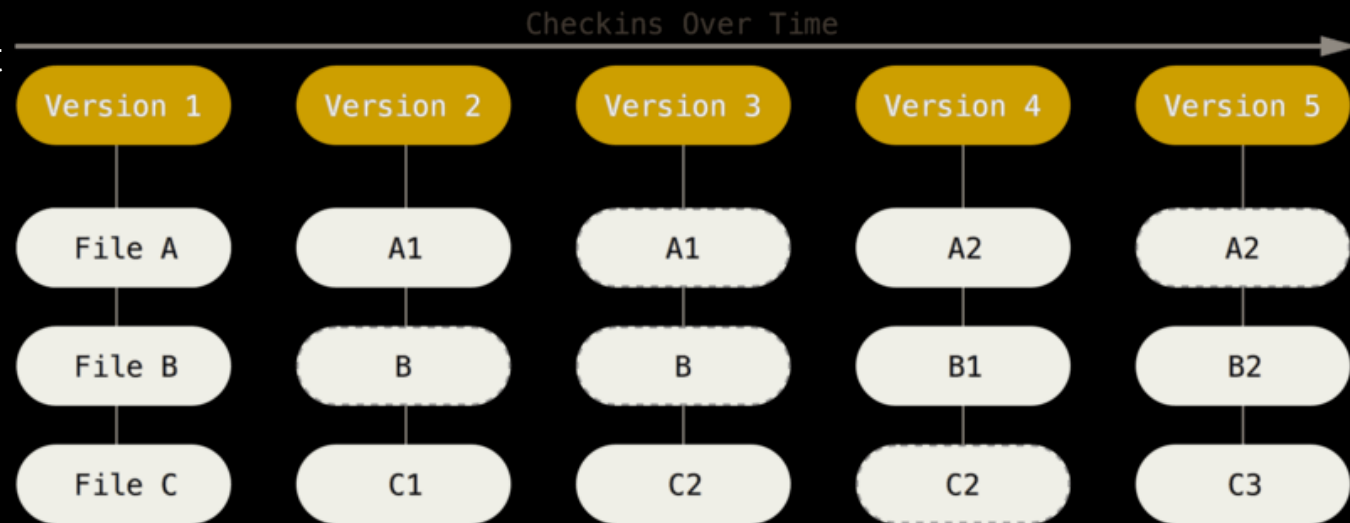
VCS Approaches

Take a linear history by creating snapshots

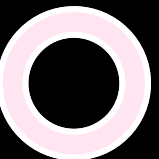


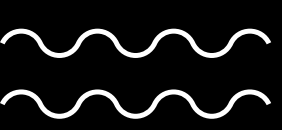
Git: snapshots of changes

- We can use git in this history module
- Can work for version control without collaborators
- Snapshots, not differences
- If a file is not changed, git store only a link to the previous version



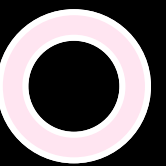
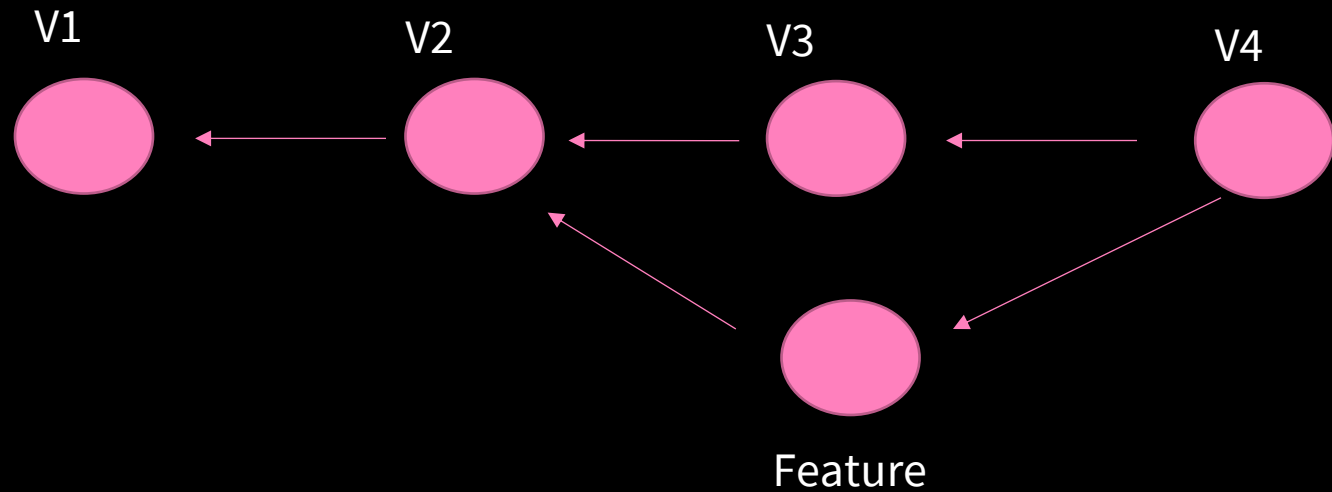
Credit: Pro Git book, Git Documentation





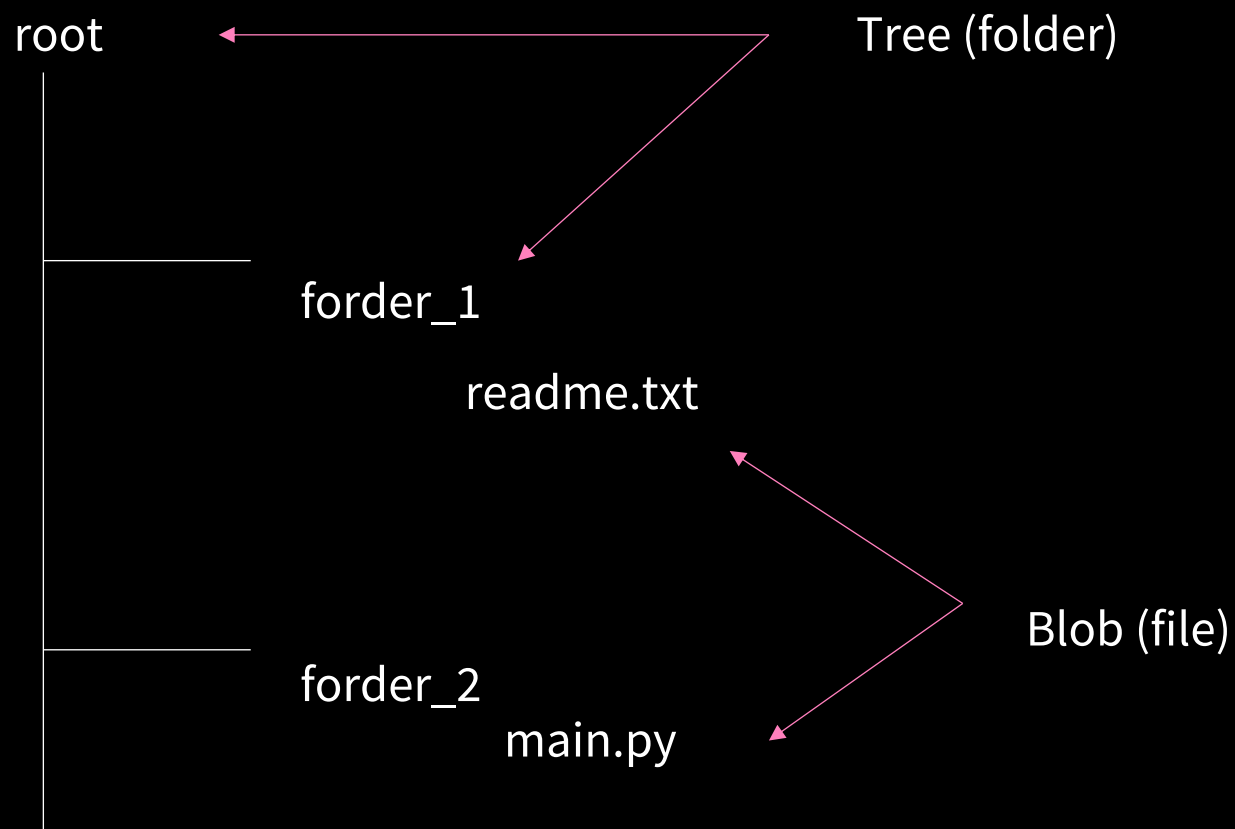
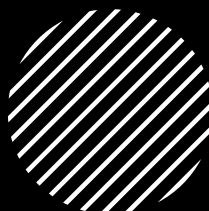
Git: history model

Git: Directed acyclic graph history



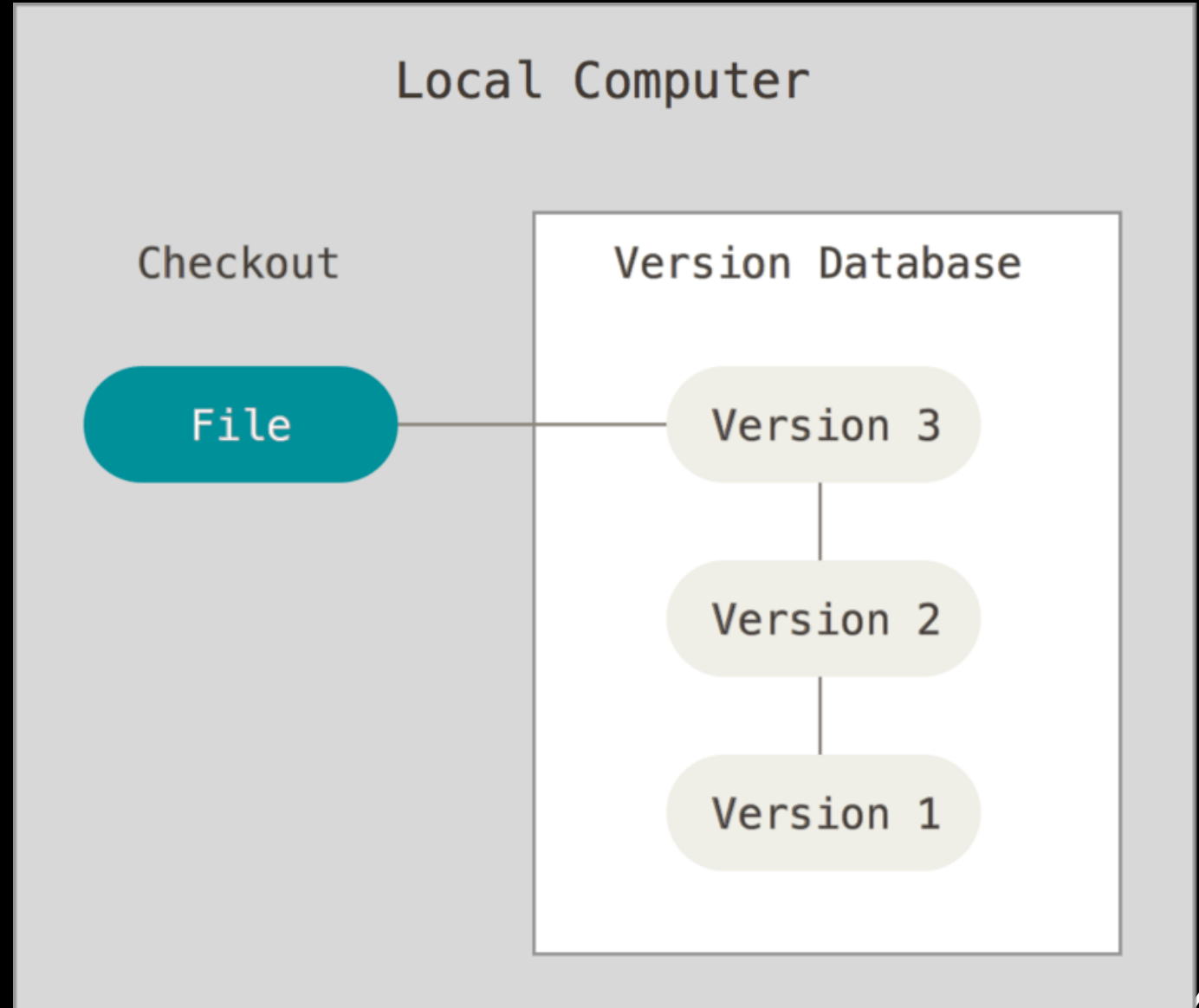


Directory structure



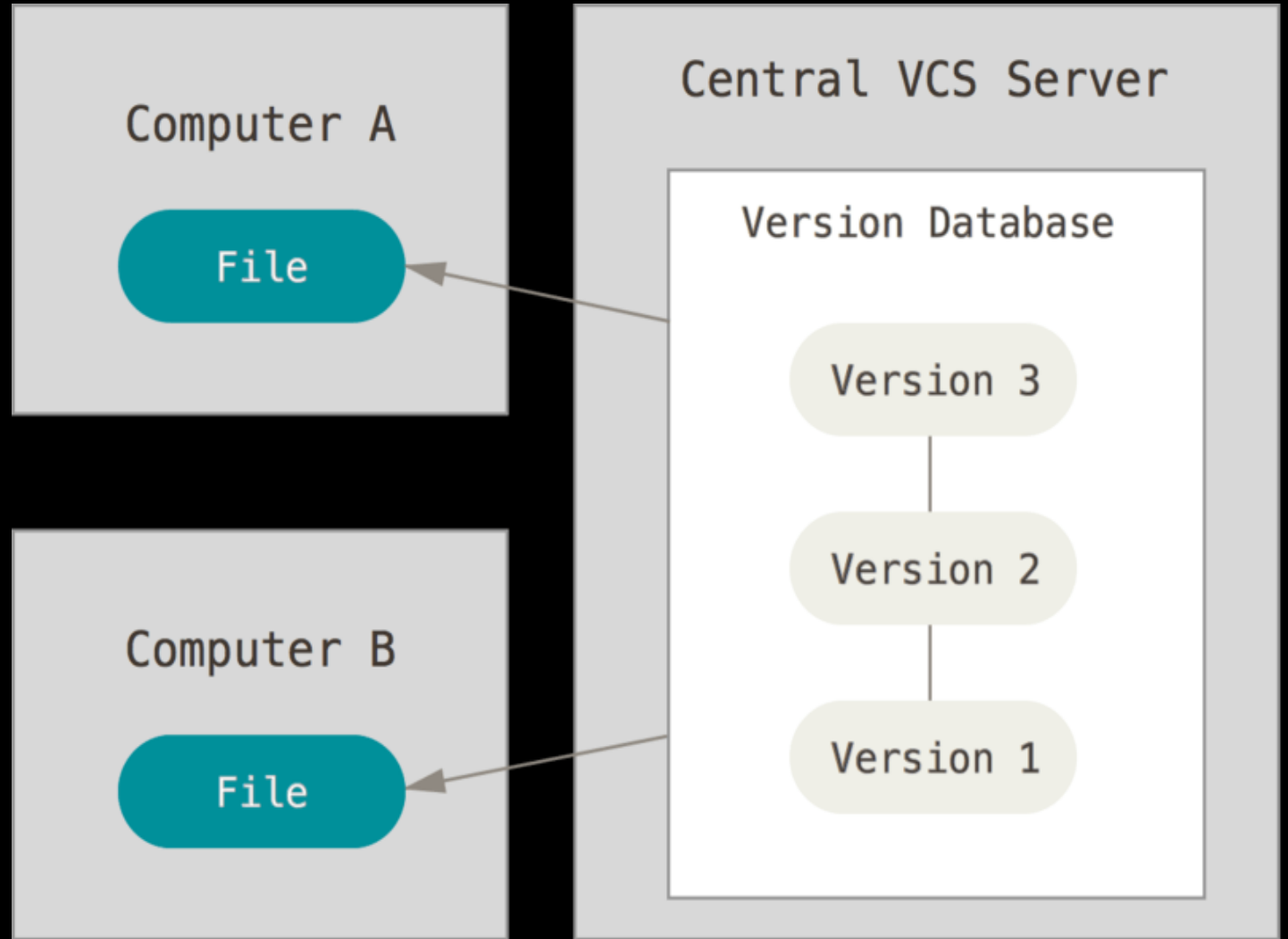
Local VCS

- Simple database
- Keep all changes to files
- RCS (Revision Control System)
- Stores path sets (differences between files)



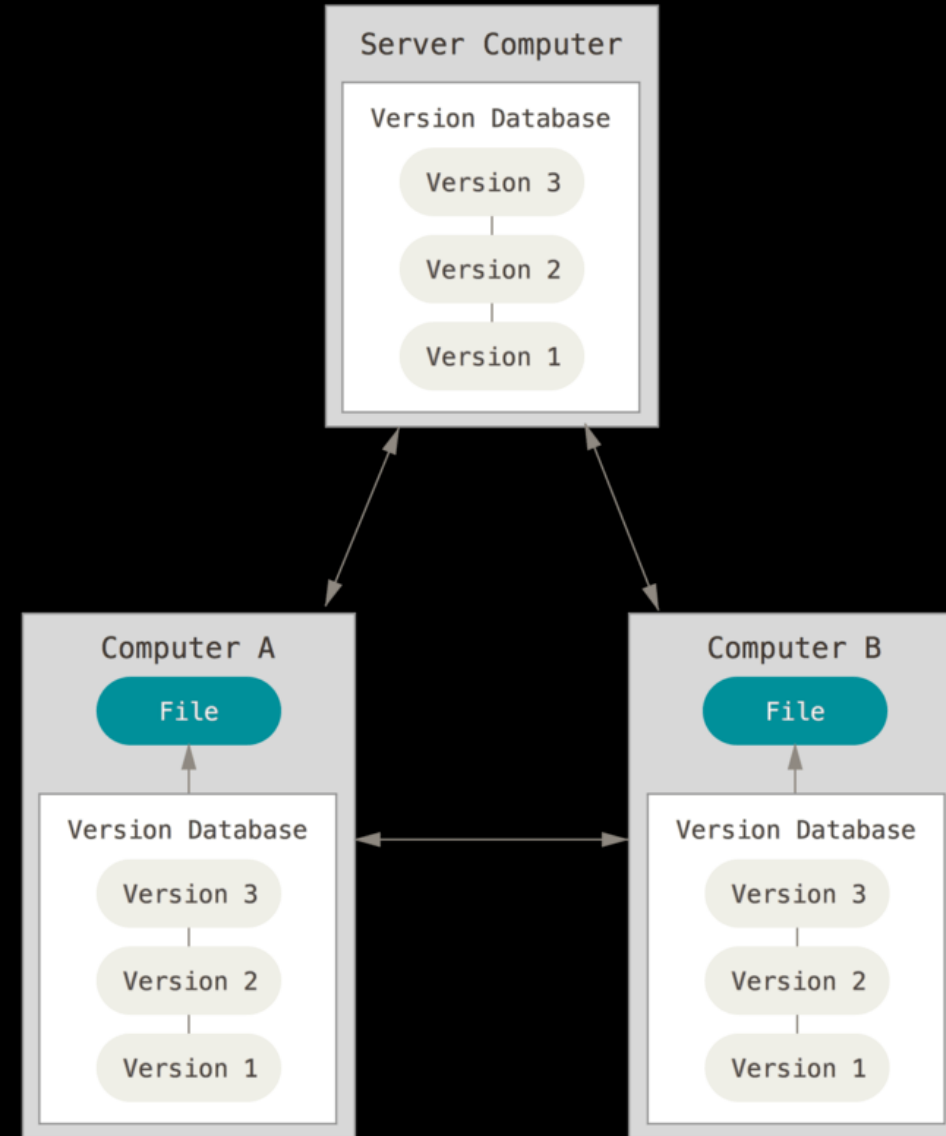
Centralized VCS

- Developers need to collaborate
- CVS, Subversion, Perforce
- Server that contains all versioned files
- Clients that check out files
- Everyone knows what everyone else is doing in a project
- Administrator control
- Why this is was not idea?



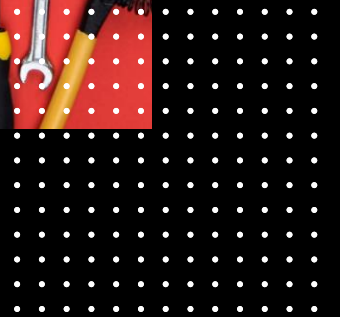
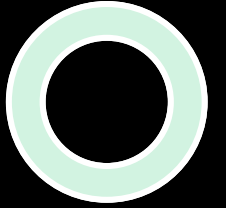
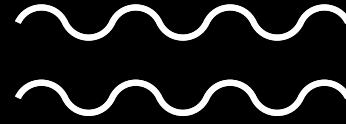
● Distributed VCS

- Git model
- Each client mirror the entire repository
- Each clone is a copy of the project
- Can deal easily with having multiple remotes



The command line with git

- Why use the CL with git?
 - 1- Can run all Git's functionalities.
 - 2- if you know how to use Git on the CL, you probably can use it in another environment.



● Useful (linux) command

pwd: Print working directory

cd: Change directory (cd /mnt/c/users/example/desktop)

cd .. : One level up

ls: «List» It lists the content of directory

ls -a: list all files and directory (includes hidden)

clear: clean the shell

cat example_file: see the content of a file

mkdir dir_name: create a new directory in your working path

man command: gives the documentation of a command

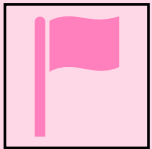
command --help: also displays information about certain command



● CL: task



List the files of a directory & sort them by size



Find the usage of `-r` flag in `rm` and use it



Use `head` or `tail` commands to display certain number of lines (5 or 7 or any number)



GIT: GETTING STARTED

FOLLOW THE
FOLLOWING [LINK](#) TO
DOWNLOAD GIT

Downloads



Older releases are available and the Git source repository is on GitHub.

Latest source Release

2.45.1

[Release Notes](#) (2024-04-29)

[Download for Windows](#)



● Git: installation

- Linux (Debian): `$ apt-get install git`
- macOS: `$ brew install git` #install homebrew if you don't have it
- Windows: GUI installer



● Git: Installation

- Check if git is installed on the CL:

```
ahmad-unibe@ahmad-unibe:~$ git --version  
git version 2.34.1
```



● Git: config

- Git needs to know who is doing changes to a repo
- Need to set config for user & email
- 3 levels of git config (system – global – local)

```
$ git config --global user.name "Ahmad"
```

```
$ git config --global user.email  
"ahmad.alhineidi@unibe.ch"
```

```
$ git config --local core.editor "nano"
```

```
$ git config --system --list
```



● Git: help (\$ git <command> help)

- Two levels of help
- Detailed man page for each command
- Short help with main functionality -h

\$ man git #manual page of git

\$ git help config #long man page

\$ git config -h # Shorter help version

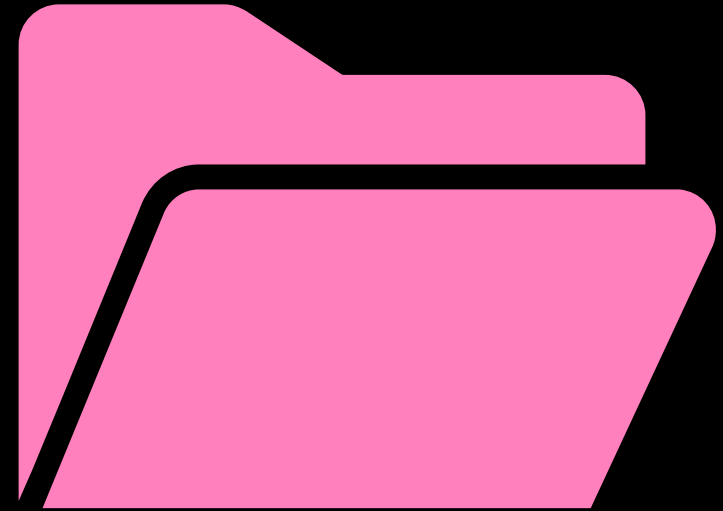




Git: start a repo

- There are two ways to have a git repo:

- 1- Start tracking a local directory with `git init`
- 2- Clone a git repo from somewhere else (`gitlab – github`)



● Git: The 3 stages cycle

- Git workflow goes through 3 stages of creating a new commit
- The stages are (modified – Staged – committed)
- Git-status: helps locate you in the process



● Git: The 3 stages cycle

- Modified (Untracked): Changes to file or directory but not added to git database.

```
ahmad-unibe@ahmad-unibe:~/git-demo$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file.txt

no changes added to commit (use "git add" and/or "git commit -a")
```



● Git: The 3 stages cycle

- Staged: The modified file has been marked for changes to be committed. git-add

```
ahmad-unibe@ahmad-unibe:~/git-demo$ git add file.txt
ahmad-unibe@ahmad-unibe:~/git-demo$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file.txt
```



● Git: The 3 stages cycle

- Committed: The changes are safely stored to your local database. `git-commit`

```
ahmad-unibe@ahmad-unibe:~/git-demo$ git commit -m "add new feature"
[master a3b7f8d] add new feature
1 file changed, 5 insertions(+), 1 deletion(-)
ahmad-unibe@ahmad-unibe:~/git-demo$ git status
On branch master
nothing to commit, working tree clean
```



● Commit: best practices

- Commit Related Changes
- Commit frequently
- Keep it short (about 50 characters)
- Don't commit half-done work
- Test your code before commit
- Use imperative voice “fix bug”, instead of “fixed bug”
- Leave second line blank if writing long commit



● Commit: best practices

- What is wrong with this staging / committing?

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

```
new file:   .DS_Store
new file:   products.ts
new file:   registration.test.ts
new file:   registration.ts
new file:   validation.test.ts
new file:   validation.ts
```

```
git commit -m 'Updated various areas such as validation,
registration and products pages'
```



● Task: Try it yourself

- Create a new directory
- Create a git repository inside the directory
- Add files into the directory
- Commit the changes of the directory into git



● .gitignore

- Sometimes we don't want to track all the files/folders in a repo, why? Think of an example
- Create a file inside your repository called .gitignore
 - with linux: `$ touch .gitignore`
 - with winows: `$ type nul > .gitignore`
- Write the name of the file inside .gitignore
- More information on .gitignore syntax: <https://git-scm.com/docs/gitignore>



● Task: .gitignore

- Add .gitignore into your repository
- Add three .txt or .py files into the repository
- Have git ignore these 3 files by writing 1 line into .gitignore
- Add a fourth txt file into your repository but make sure it doesn't get ignored by git



● Git-log: show history

- To list the commits made in a repository use the command log
\$ git log
- log shows all the commits in a reverse chronological order
- Discover the git log manual page and find some useful options to use with.



● Git: undoing changes

- Git allows you to go back one step backward in the 3 stages cycle
- Commands used to undo changes (git-checkout – git restore – git reset)



● Git: undoing changes

- Reverse changes to modified file until last commit
- `$ git checkout -- <file_name>`
- `$ git restore <file_name>`
- Use with CAUTION!



● Git: undoing changes

- Un-staging a staged file
- `$ git restore --staged <file_name>`
- `$ git reset HEAD <file_name>` #older git versions
- Why would this be useful?



● Git: undoing changes

- You can edit a commit by using the argument --amend
- `$ git commit --amend`
- This will allow you to edit a commit. It gives the possibility to stage other files and add them to the commit.



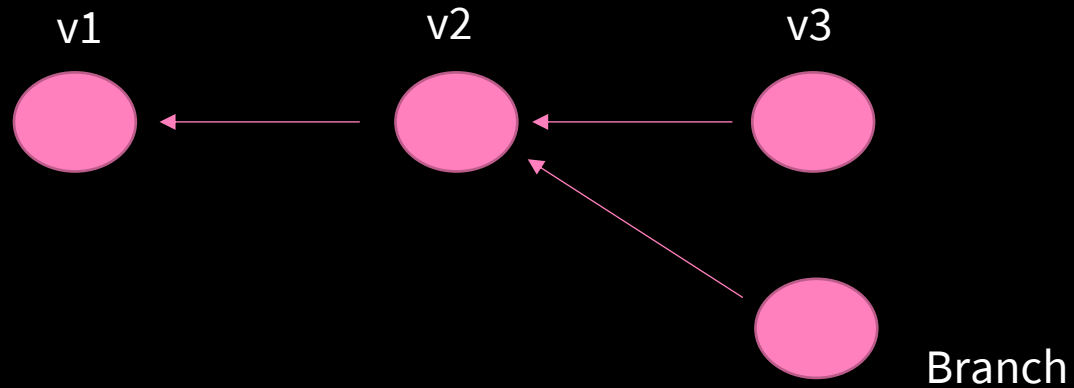
● Task

- Make changes to a file in your git repository
- Undo these changes until the last commit
- Make several changes and corresponding commits to file then undo these changes to a certain commit (not last one)





Git: Branching



- A pointer to one of your commits
- Most VCS have branching support
- Git is unique in handling branching (Less expensive, lightweight)



● Git: Branching

- Check the branch name of your repository:
\$ git branch #by default, called master or main
- Create new branch:
\$ git branch <Branch name>
- See the last commit in each branch
\$ git branch -v
- Move to work in a certain branch:
\$ git checkout <Branch name>
- Git does not switch to new branch automatically after creating it!



● Branch history: git-log

- Git log will only show the commits history for the main or master branch when you are in it. To show commits in other branches:

```
$ git log <branch_name>
```

or

```
$ git log --all
```

- Use the argument --oneline to display less information when having too many commits:

```
$ git log --all --oneline
```



● Git-switch

- Newer versions of git (2.23) uses switch command to instead of checkout:

```
$ git switch branch_name
```

- Delete a git branch:

```
$ git branch -d branch_name
```

```
# -D if there are unmerged elements
```



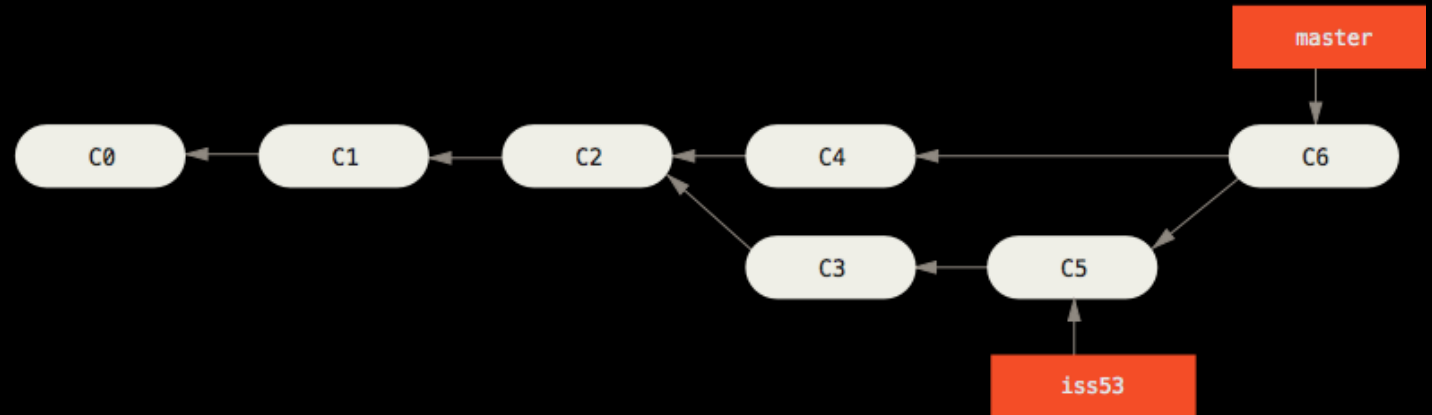
● Discussion & task

- Why branching is useful?
- Create a new branch and switch to it in one command (there are 2 possible ways)
- Do changes in that branch then try to delete it



● Git-merge

- Eventually you want to merge changes into your main or deployment branch
- Git-merge command
- Merge conflict can occur
- Why does a conflict happen?





Git-merge

- Merging branch: Go to the branch you want to merge into:

```
$ git merge <branch_name>
```

- Dealing with merge conflict:

```
$ git mergetool OR manually
```

- See which branch is merged and not merged into your current branch

```
$ git branch --merged
```

```
$ git branch --no-merged
```

- Abort a merge: Sometimes when you have a merge conflict, you want to abort the merge and fix it in a different way. This is done with the abort flag:

```
$ git merge --abort
```

- An option would be to go back to the last commit:

```
$ git reset --hard HEAD #use with caution. All changes until last commit are deleted!
```



● Task

- Create new branch and do changes in it and commit them.
- Do changes in your master branch to the same file and the same lines
- Try to merge the branch and resolve the conflict



● Merge conflict strategies

- Many ways to deal with conflicts
- Example:
 - X ours: to accept changes in current branch
 - X theirs: to accept changes in the merged branch



● Git-aliases

- For a faster & more sufficient workflow use local aliases
- Examples from Pro git book:

```
$ git config --global alias.co checkout  
$ git config --global alias.br branch  
$ git config --global alias.ci commit  
$ git config --global alias.st status
```



● Git-gui

- Git usually comes with graphical user interface.
\$ git gui
- Most IDE comes with built-in functionalities to use git

Task: Try doing changes and committing then using the IDE you use



● Github

- What is github and how is it different from git?



● Github

- Largest online host for git repositories
- Central point for collaboration for millions of developer
- Many open-source projects on Github
- Owned by Microsoft (Not part of the Git open-source project)



● Github: usage

- Interaction with local git repository
- Github Pages: Static web hosting service
- Gist: code snippets
- Continuous integration/continuous deployment (CI/CD)
- Other features



● SSH connection

- Secure shell is used to connect your local repo into github
- Your shell comes already with SSH-agent
- Follow the instruction here to create new ssh key and add it to github [link](#)
- Once set up, you will be able to pull/push code from/to github



● Git: working with remote

- Clone your new repository into your local device either with https or ssh:
\$ git clone git@github.com:<user name>/<repo name>.git
- You can list the name Git gives to the remote repository that you cloned:
\$ git remote #This will give you at least «origin»
- You can also view the URLs which git specify to reading and writing to the remote repo:
\$ git remote -v
- Add changes to your remote repository that you did locally:
\$ git push
- Pull changes that were made to the remote repository:
\$ git pull



● Git/Github: main functionalities

- Git-clone: clones “copies” a repository into a new directory
\$ git clone <source>



● Git/Github: main functionalities

- Fork: You can fork a project in github into your private account to work on it. This is usually is done when you do not have push access to that project.
- Once you've done some changes, you can do a pull request
- The original repo owner can approve and merge



● Github: adding collaborators

- You can add collaborators to your repository on github to work with others.
- Is done from settings inside the repository --> collaborators --> add people



● Github pages

- Free static website host
- Demo



● Teamwork

- Group in teams of 2 or 3 and choose a task from the following:

[Task one \(Python\)](#)

[Task two \(Python\)](#)

[Task three \(R\)](#)

[Task four \(R\)](#)

[Task five \(github pages\)](#)

