Quick references for DSA 104 participants (who haven't worked with git / GitHub before)

Aim: Core understanding of git and GitHub – but mainly as user. In DSA104 we only use very few functions of the git workflow.

Resources:

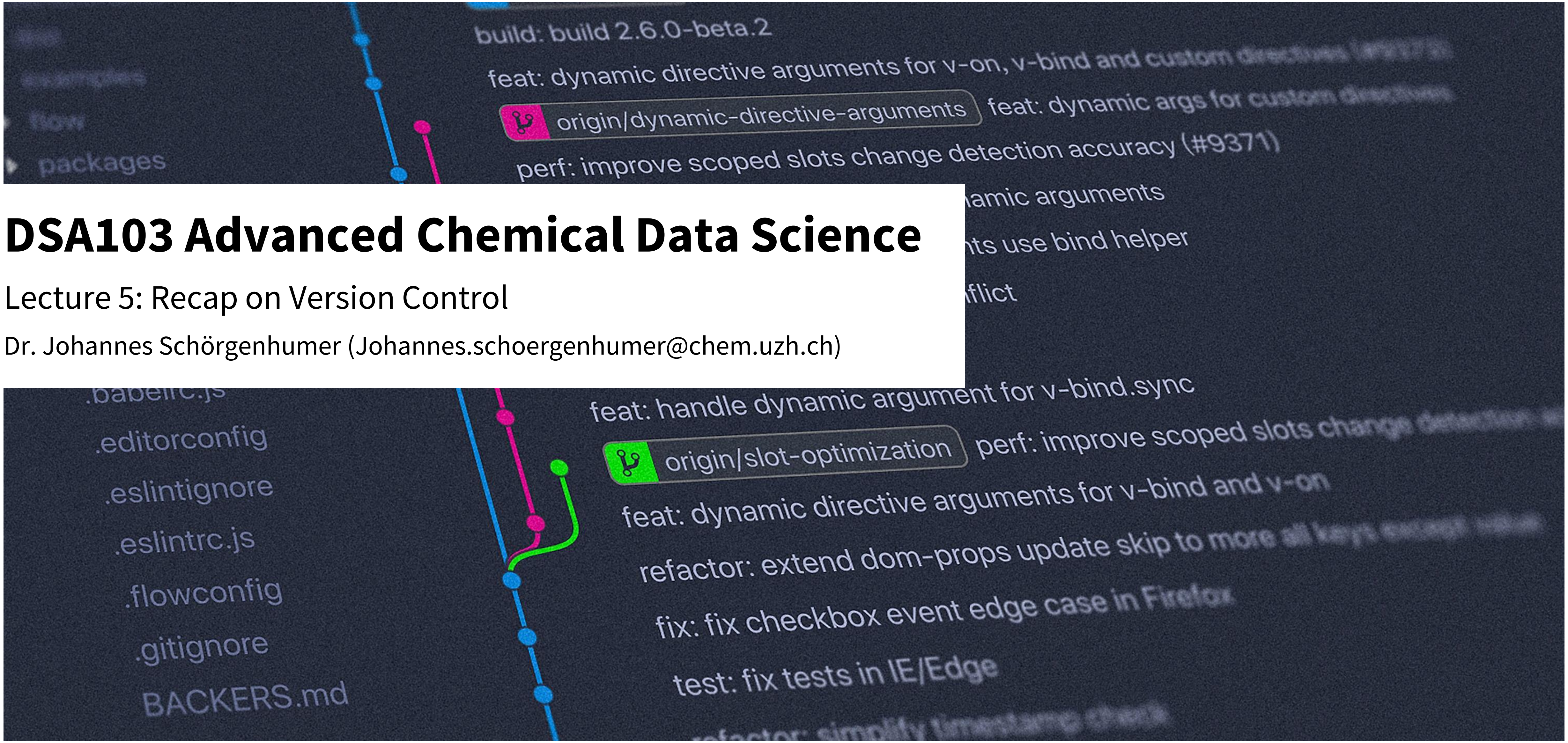
- Summary in the following lecture slides from DSA 103 lecture 5.
- Online tutorials, e.g.
  - https://www.w3schools.com/git/default.asp
  - https://www.geeksforgeeks.org/git/git-tutorial/
- Plenty of youtube tutorials as well.

Tasks for DSA104:

1) Fork DSA104 repo (and keep synchronized)
2) Clone your forked repo to your PC
3) Copy the dht-notebook and data to the user folder, activate virtual environment, run notetbook
4) Commit & Push changes to your forked DSA104 repo

These steps will be essentially all you will have to do in git for this course. If all of them work (including running the jupyter file), you should be all set!

Universität Zürich^UZH | Department of Chemistry



DSA103 Advanced Chemical Data Science

Lecture 5: Recap on Version Control

Dr. Johannes Schörgenhumer (Johannes.schoergenhumer@chem.uzh.ch)

Content overview

Motivation: Manual version control

What are VCS? (Benefits, Types, examples)

Tools: Git vs. GitHub vs. GitLab

How git works – principles, terminology, workflows

Activity (Set up gut, Explore basic functionality)

Branches and Merging

Exercise (branches, merging, conflicts)

Git: Best practices

GitHub: Overview

Motivation: Manual Version Control

Imagine working on your thesis…

- Thesis.docx

When you have most of the text in, you try some formats:

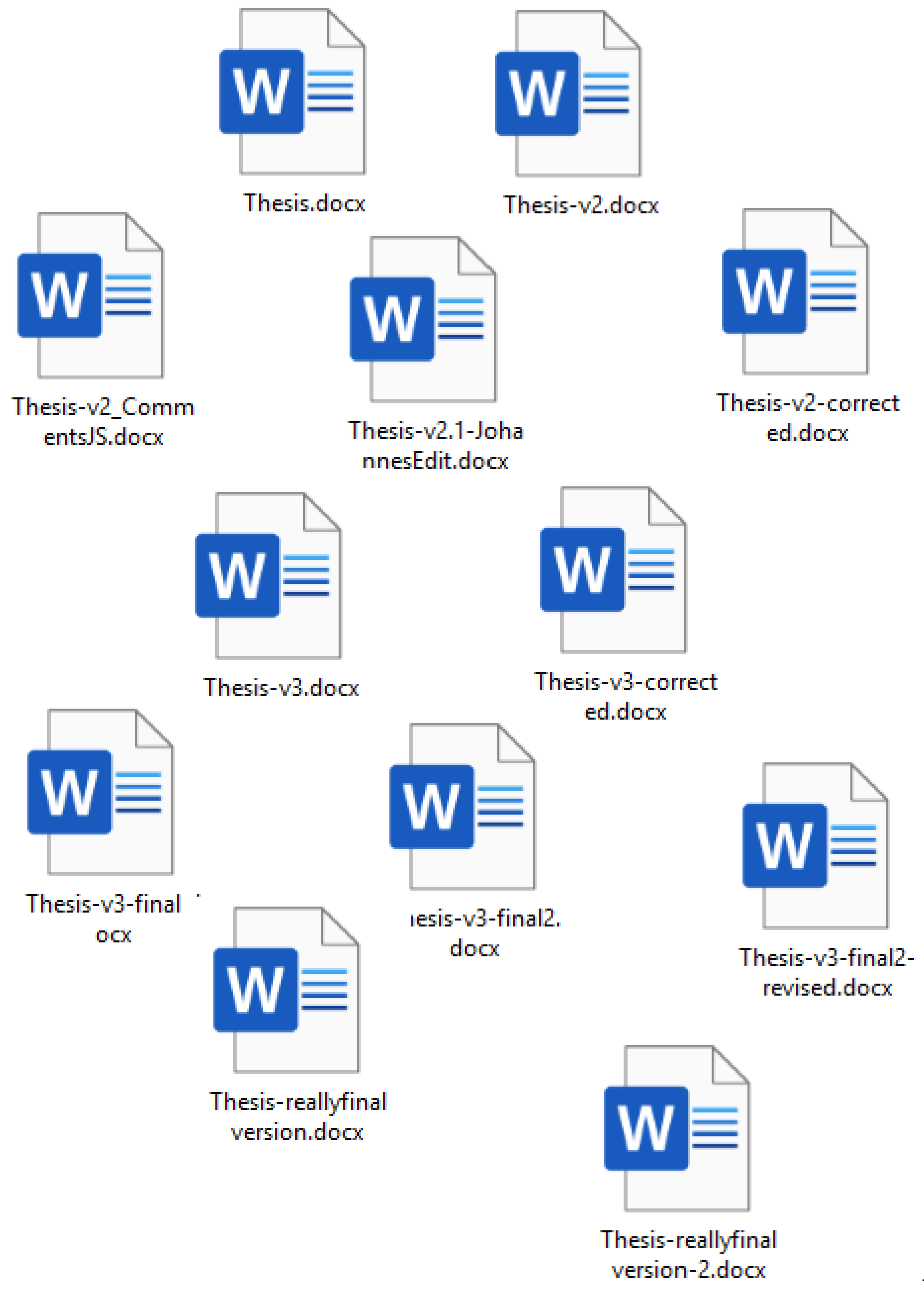
- Thesis-v2.docx

Then you send it to some colleagues for proof reading:

- Thesis-v2_CommentsJS.docx
- Thesis-v2.1-JohannesEdit.docx
- Thesis-v2-corrected.docx

You consolidate all comments:

- Thesis-v3.docx
- …


Thesis.docx
Thesis-v2.docx
Thesis-v2_CommentsJS.docx
Thesis-v2.1-JohannesEdit.docx
Thesis-v2-corrected.docx
Thesis-v3.docx
Thesis-v3-corrected.docx
Thesis-v3-final ocx
esis-v3-final2.docx
Thesis-v3-final2-revised.docx
Thesis-reallyfinal version.docx
Thesis-reallyfinal version-2.docx

Version Control Systems (VCS)

Benefits of VCS:

- Tracking of changes without keeping multiple files
- Long-term time change history: Hopping between different timepoints in development
- Branching out for testing, merging for integration and revert changes if unsuccessful
- Traceability: Changes and contributors

But there is more in the bigger picture:

- Co-development efficiency
- Risk management (safety net)
- Transparency
- Improve quality
- Enable integration of code / data
- Enable automation of tasks

What are Version Control Systems (VCS)

VCS are indispensable tools for coding and the backbone of collaborative projects in the field of programming and data science

- Concept was introduced early on: precursors in 60s, IBM’s Source Code Control System in 1972
- Landscape was revolutionized in 2005 with the introduction of git by Linus Torvalds
- Many different tools emerged to deal with specific development requirements (git is not the only example)
- Examples: Git, Subversion(SVN), Mercurial, Azure DevOps Server, …

Linus Torvalds (2002) - Linuxmag.com, Dezember 2002, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=17991

What are Version Control Systems (VCS)

Types:

- Centralized VCS (e.g. SVN):
  - All versions are saved on a centralized server
  - All changes are committed and pushed directly to the server
  - Ease of management, but no offline coding possible

- Distributed VCS (e.g. Git):
  - Every user has a complete copy of the repository
  - Working offline is possible
  - Robust framework for merging and branching
  - Improves efficiency and redundancy

Git vs. GitLab vs. GitHub

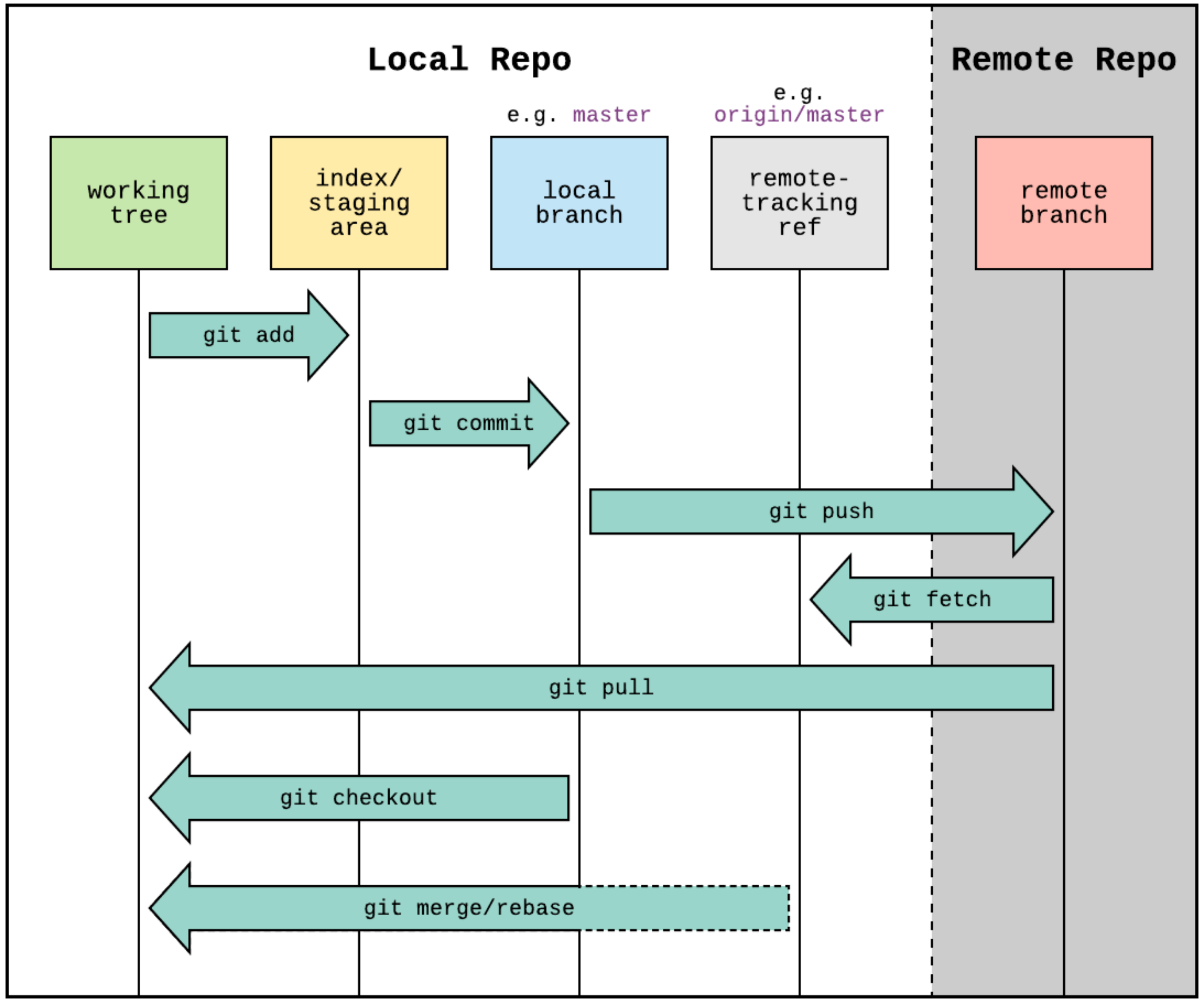
Git: Opensource VCS, basis for other tools:

GitLab: web-based platform to streamline development workflows. Includes git repository management and combines with continuous integration/continuous development (CI/CD) to automate entire delivery process

GitHub: Cloud-based hosting service for managing git repositories, focused on sharing with collaborators or making codebase available publicly (Social coding)

We will be using GitHub! (arguably most common platform for sharing data in scientific community) for this you need to understand the basics of git…

How Git works - Git basics



Cheat sheet: https://education.github.com/git-cheat-sheet-education.pdf

- Repository/"Repo": digital storage for project files and complete change history (tracked commits)
- Commit: Snapshot of project at specific point in time, any changes are tracked
- Branch: Parallel version of your project to work on different features or fixes independently
- Checkout: Switching between branches or specific commits in your repository
- Merge: Combining changes from one branch into another, usually to integrate new features or fixes
- Conflict: A situation where Git cannot automatically merge changes due to overlapping edits
- Fork: personal copy of someone else's repo, changes do not affect the original repo (without pull request)
- Clone: Creates a local copy of a remote repository
- Pull: Fetching and integrating changes from a remote repository into your local repository
- Push: Sending your committed changes from your local repository to a remote repository

Activity: Git and Local Repositories

Install git, using git in IDE

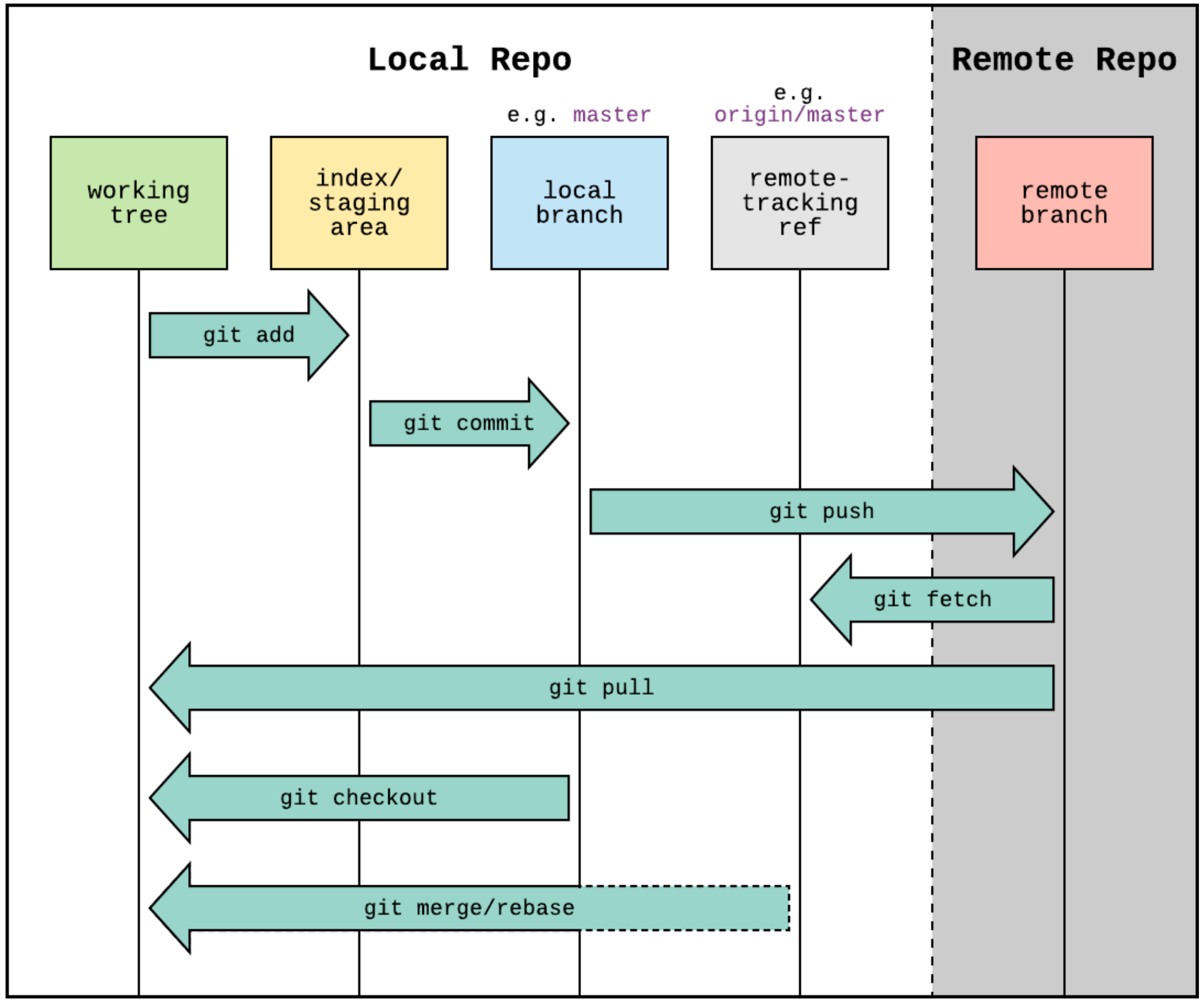
Initialise a local repository

Write a first script

Add to staging area

Commit changes

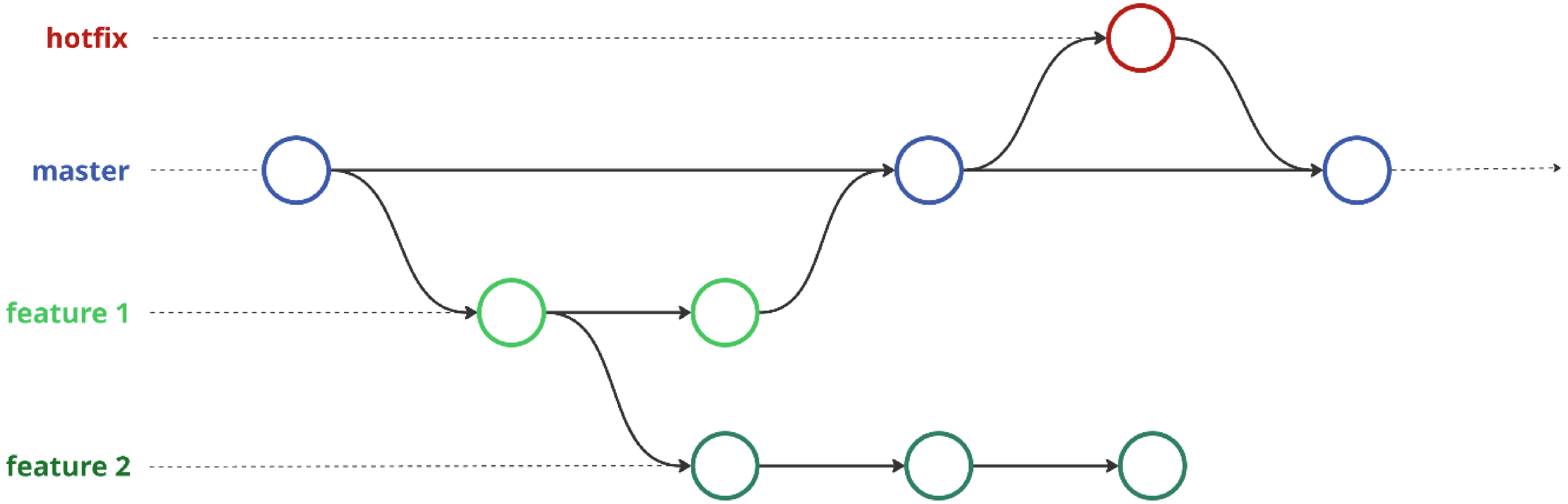
How Git works - Git basics



Cheat sheet: https://education.github.com/git-cheat-sheet-education.pdf

- Repository/"Repo": digital storage for project files and complete change history (tracked commits)
- Commit: Snapshot of project at specific point in time, any changes are tracked
- Branch: Parallel version of your project to work on different features or fixes independently
- Checkout: Switching between branches or specific commits in your repository
- Merge: Combining changes from one branch into another, usually to integrate new features or fixes
- Conflict: A situation where Git cannot automatically merge changes due to overlapping edits
- Fork: personal copy of someone else's repo, changes do not affect the original repo (without pull request)
- Clone: Creates a local copy of a remote repository
- Pull: Fetching and integrating changes from a remote repository into your local repository
- Push: Sending your committed changes from your local repository to a remote repository

Branches and Merging

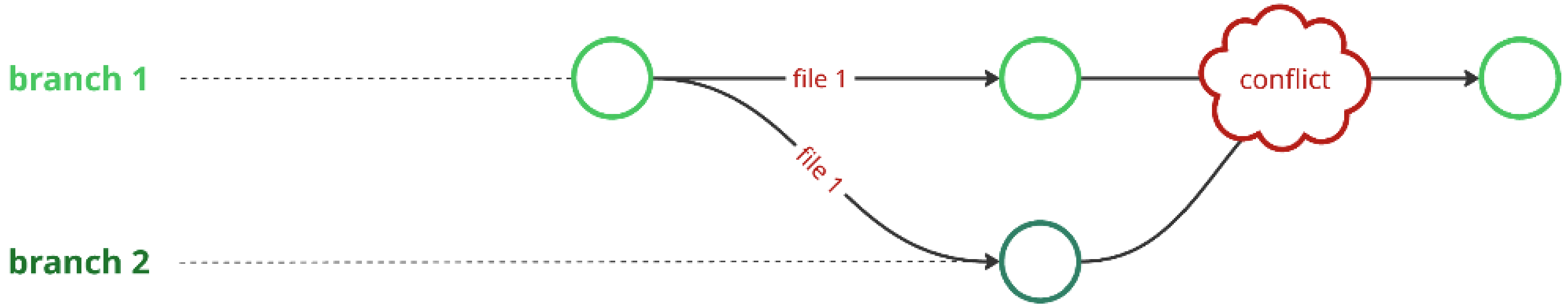


Branches and Merging

Merge conflicts arise if the same file is changed at the same time.

Need to be resolved manually.

Best to avoid by clearly structured branches.



Activity: Branches

Create a branch in your local repository

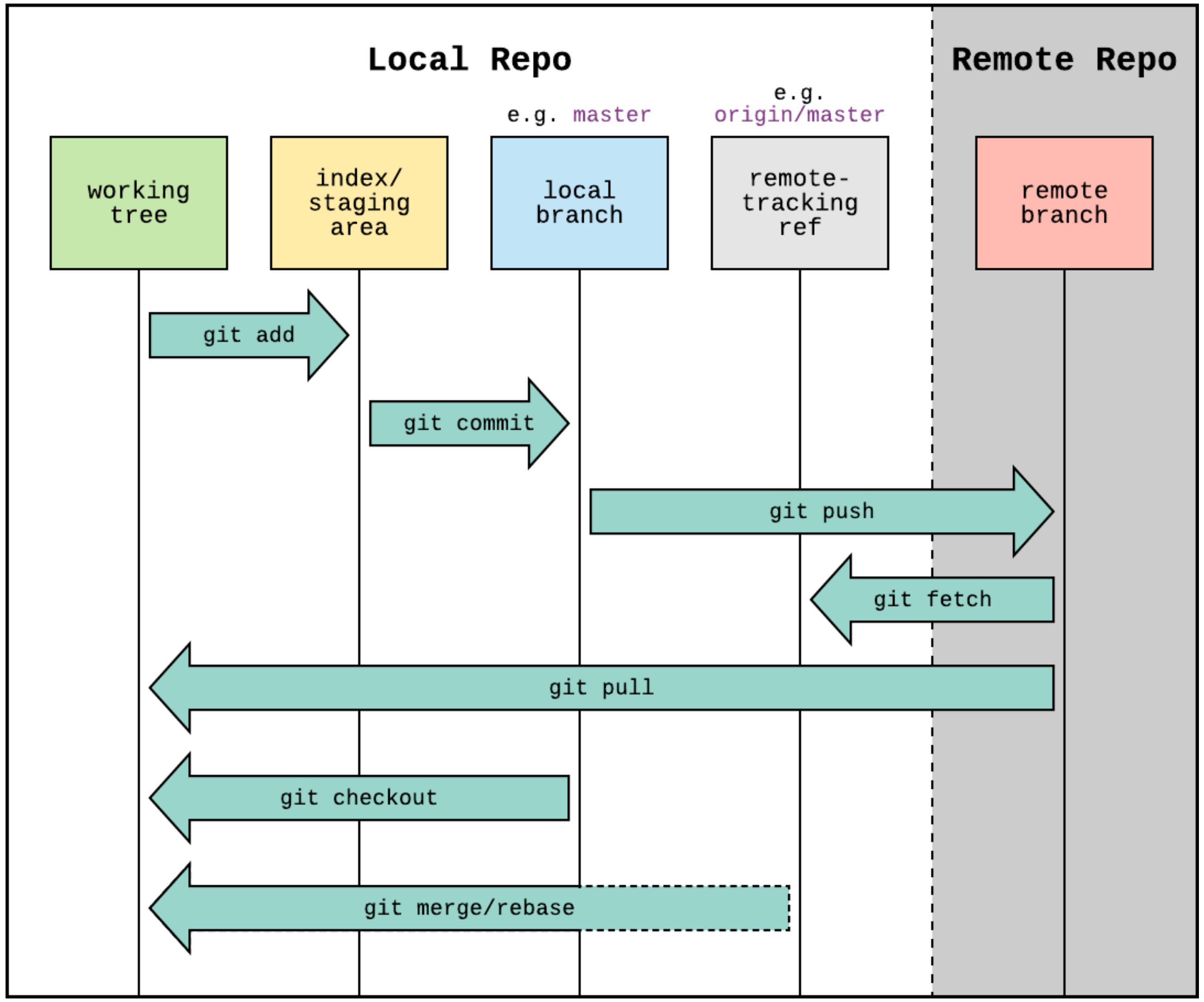
Merge the development branch into the master branch

Create another branch

Create a merge conflict: modify the same file and commit on two different branches

Resolve merge conflicts

How Git works - Git basics



Cheat sheet: https://education.github.com/git-cheat-sheet-education.pdf

- Repository/"Repo": digital storage for project files and complete change history (tracked commits)
- Commit: Snapshot of project at specific point in time, any changes are tracked
- Branch: Parallel version of your project to work on different features or fixes independently
- Checkout: Switching between branches or specific commits in your repository
- Merge: Combining changes from one branch into another, usually to integrate new features or fixes
- Conflict: A situation where Git cannot automatically merge changes due to overlapping edits
- Fork: personal copy of someone else's repo, changes do not affect the original repo (without pull request)
- Clone: Creates a local copy of a remote repository
- Pull: Fetching and integrating changes from a remote repository into your local repository
- Push: Sending your committed changes from your local repository to a remote repository

Activity: Remote Repository

Register on GitHub

Overview on GitHub: Features, Management, …

How to create a remote repository on GitHub

Fork and clone a public repository

Best practices

The repository setup: Define environment, access control (Templates)

Establish workflow alignment: Define coworking rules

Write clear commit messages: Explain the changes in short and concisely. Use standardized nomenclature (e.g. conventionalcommits)

Commit often: Small commits, concise messages.

Test before committing: Use linters (code formatting!) and testing tools (e.g. pytest) and CI workflows to do that automatically

Commit only when ready

Avoid conflicts: Always use up-to-date code, make use of branches!

Use branches: Code branches support multiple versions of software releases and patches.

Limit repository access: Only to contributors who need it, control unauthorized access

https://github.com/resources/articles/software-development/what-is-version-control

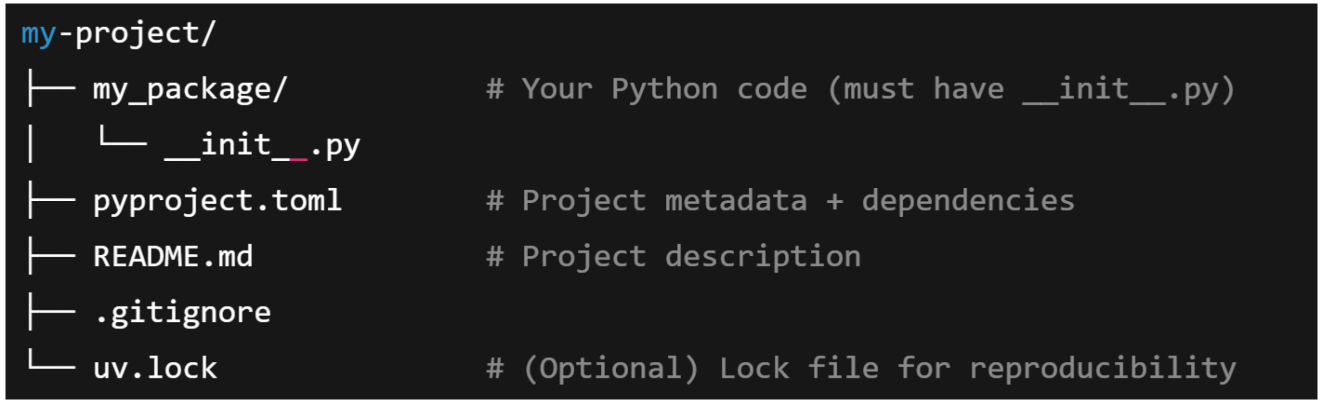
Best practices – useful tools

Project templates (cookiecutters): Provide standardised and consistent environment for working on a project, many different templates available, *e.g.* for python scripts, Django or data science (https://www.cookiecutter.io/)

Package managers:

- Help to **define, lock, install, and reproduce** project dependencies reliably and efficiently. Examples: pip, uv, conda
- Key aspects: **dependency management** (auto-install needed packages), **lockfiles** (exact versions of dependencies ensure reproducibility), **virtual environments** (isolation from system environment), **installability** (lets others install your package with one command, *e.g.* uv pip install mypackage)

Installable repos / packages: follow clear data structure (*vide infra*), include .toml file to declare dependencies and a lockfile to specify versions. Voilà!



```
my-project/
├── my_package/           # Your Python code (must have __init__.py)
│   └── __init__.py
├── pyproject.toml        # Project metadata + dependencies
├── README.md             # Project description
├── .gitignore
└── uv.lock               # (Optional) Lock file for reproducibility
```

Linters: Analyse code for style errors, bad practices and style violations (*e.g.* flake8, ruff)
Formatters: Rewrite / modify code to follow consistent style such as line breaks, etc. (*e.g.* ruff)

Exercise preview

https://github.com/schoergj/DSA103/tree/main/exercises/lecture_5