Git

Clemens Schmid

The command line

The command line

A text-based interface to control your computer

- Born in the 70s to interact with mainframe systems
- An efficient, direct way to interact with files and software
- Command line software ideally engages in a dialogue with the user: https://clig.dev

Each operating sytem offers different shells and terminal emulators

- Linux shells: sh, bash, ksh, zsh, . . .
- Linux terminal emulators: XFCE/GNOME terminal, Konsole, alacritty, kitty, terminator, . . .

Windows users in this workshop should use the Git BASH, which simulates a Linux environment. Please use “Run as an administrator” (apparently necessary)

Navigating

Show your current position relative to the root directory

```
pwd
```

Show the content of a directory

```
ls
ls -l # long format output
ls -h # file sizes in human readable format
ls -a # also show hidden files
```

Move to another position in the file system

```
cd path/where/I/want/to/go
cd / # move to the root directory
     # absolute paths start with `/`
cd ~ # move to your home directory
cd .. # move one level up to the parent directory
```

Creating and editing files

Print text and forward it to a new file

```
echo "test" # Print a value to the command line
echo "test1" > file.txt # write text to a (new) file
echo "test2" >> file.txt # append text to a file
```

Edit a file with a minimal text editor

```
nano file.txt
```

- `Ctrl`+ `x` to close nano
- Cut & paste: `Ctrl` + `Shift` + `C` & `Ctrl` + `Shift` + `V`
- There are more fancy command line text editors (emacs, vi)

Copying, moving and deleting files

Copy a file

```
cp file1.txt file2.txt
```

Move and/or rename a file

```
mv file1.txt file2.txt
```

Delete a file

```
rm file2.txt
```

Making and deleting directories

Create a directory

```
mkdir myDir
```

Copying and moving directories works just as for files with `cp` and `mv`

Delete a directory

```
rm -r myDir
```

The `-r` (“recursive”) flag is necessary to delete a directory

Looking up features

The most important tools have extensive manuals

```
man ls # man + name of the program
```

```
NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information about the FILEs
       (the current directory by default).
       Sort entries alphabetically if none of
       -cftuvSUX nor --sort is specified.
...
```

Modern command line tools

Modern CLI tools are structured as a dialogue between you and the computer: You don't have to remember details

```
git # shows an overview of the important subcommands
git commit -h # shows the options for one subcommand
```

```
usage: git commit [<options>] [--] <pathspec>...

    -q, --quiet           suppress summary after commit
    -v, --verbose         show diff in commit message template

Commit message options
...
    -m, --message <message>      commit message
...
```

What is Git?

The software

- https://git-scm.com
- A free and open source distributed version control system
- Written by Linus Torvalds 2005 (an initial version in only 3 days!)
- Developed since then, now v2.37.1
- "Git" has no clear meaning: *Global information tracker*, *Goddamn idiotic truckload of sh*t*

Features:

- Fast (logging changes almost instantly)
- Robust (Almost never breaks, if used correctly)
- Scales incredibly well (small to very large projects)
- Is relatively easy to use and an almost universal standard

Version control

Traditional version control

- Many iterations of one file:
  - Manuscript.doc
  - Manuscript_revised.doc
  - ...
  - Manuscript_final35.doc
  - ...
- Collaboration hell:
  - Sending the manuscript back and forth via email
  - Manually merging contributions

Google Docs etc. solved some of these problems for text. But what about data and code?

Version control with Git

- One iteration of said file:
  - Manuscript.md
  - an unintrusive log of iterative change
  - the option to jump back to any previous version
- Collaboration as a first-class citizen:
  - decentralized data keeping and backup
  - direct and fair documentation
  - tools to resolve merge conflicts

Git handles text, code and (small) datasets

Version control

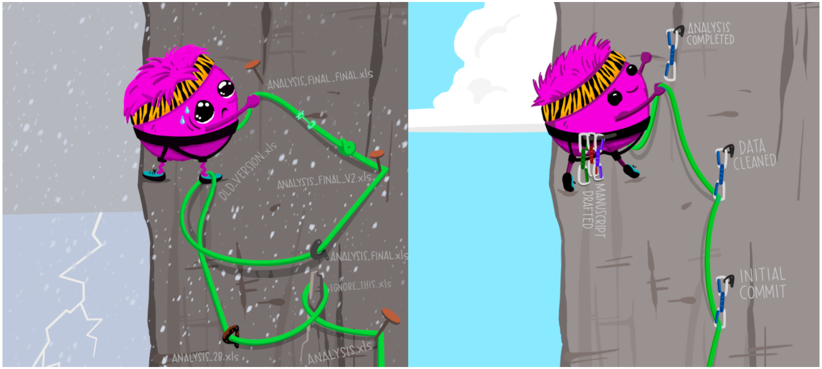

Figure 1: Git helps to streamline your work (© Allison Horst and Julie Lowndes)

Git: Mechanism and terminology

Git maintains a hidden directory (`.git`) in your project directory, which documents the history and state of your project

- When you edit a file, Git automatically detects the change
- You then **add** the change to a **staging area** for logging
- When you accumulated a meaningful set of changes, you log it as a **commit** with a descriptive message
- Commits exist on **branches** of the "development tree" of your project
- The main branch is called **master** or **main** and your and your collaborators work is centered around it
- The current state of a repository can be **pushed** to a **remote** server, from where it can also be **cloned** and **pulled**

Git: An example

- Alice has a project directory, where Git is activated
- She writes an R script in a file `analysis.R`
- She **adds** the file to the **staging area** and then **commits** it with the message: *first draft of my analysis*
- Then she **pushes** the change to the **remote** on GitHub
- Her colleague Bob **pulls** the change to his **clone** of the project
- He applies changes to Alice's script, **commits**, and **pushes** again
- Alice can again **pull** the latest version of the script to develop it further

Below we will go through the details of the local portion of this workflow

Where and how to get help

Git's community is incredibly large and every reasonable (user) question is answered

- Stackoverflow features thousands of questions regarding Git, many with excellent answers
- The Git website has good documentation and tutorials (https://git-scm.com/doc)
- The Git user group is the recommended place for beginners to ask questions: https://groups.google.com/g/git-users

Git is not an obscure tool, but a central foundation of modern technology

Git in action

Running Git the first time

```
git config --global user.name "Your name here"
git config --global user.email "your_email@example.com"
```

Set up Git with your identity

- Especially relevant for later online use
- `user.name` should not be your GitHub user name, but ideally your real name
- `user.email` should ideally be the same you used for GitHub

Creating a Git repository

```
cd c: # windows only
mkdir git
cd git
mkdir myProject
cd myProject
```

```
git init
```

Initialize a directory to be managed by Git

- Creates a hidden `.git` directory, which Git uses to manage everything for this project
- Actually we rarely start like this – we rather create project on GitHub and clone that (see below)

Git's status

```
git status
```

Show the status of a project

- In a new project without any changes Git could detect, we get the following message:

```
On branch master

No commits yet

nothing to commit
```

- We're on the master branch
- We haven't added any commits yet
- There's nothing to commit, because we haven't made any changes yet

This will change – we will call `git status` frequently

Logging progress: Preparing changes

```
echo "This is a test file" > testfile.txt
ls
cat testfile.txt
```

```
git add <file>
```

Add files to the set of changes prepared for the next commit

Useful options:

- `--all`: Add all changes to the staging area
- `-f`: Force a file to be added that would otherwise be ignored (see below)

Logging progress: The commit

`git commit`

Log a meaningful set of changes

- `-m "my message"`: Every commit should be described with a clear, concise help message
- The human component – you decide how often you commit and how descriptive your commit messages are

Useful options:

- `--amend`: Add sth. or change the previous commit (only use, when you have not pushed!)

Inspection: The log

```
git log
```

See the log of past commits

```
git show <object>
```

- Every commit has
  - A unique identifier, a “hash”
  - An author
  - A timestamp
  - The message
  - The respective set of changes

Inspect Git objects, e.g. commits

```
git show <commit>
git show 15dd154496de68a9d15a4b66282650eed1390974
git show 15dd # the shortest unique string is enough
```

Inspection: Differences

```
git diff
```

See the concrete changes between two states of the Git project

- `git diff` shows the current, unstaged changes in relation to the last commit (`HEAD`) on the current branch
- Can also be used to compare different states:

```
git diff <commit1> <commit2>
```

Cleaning the staging area

`git reset`

Remove changes from the staging area

- Will not change the files, only their status in Git
- `git reset` sets everything back, but we can also apply this to individual files: `git reset <file>`

Cleaning up unwanted changes

Often we don't want to keep the changes we just made at all, because they didn't turn out to be useful or were just experiments

```
git reset --hard # set all files back
git restore <file> # set one file back
```

Reset all modifications in files already tracked by Git

- `git restore` is only available since Git v2.23, users with older versions have to use `git checkout`

```
git clean
```

Remove newly added files not yet tracked by Git

- `-d` allows `clean` to also remove entire new directories

Both `reset` and `clean` go back to the state of the current `HEAD`. Running them is irreversible!

Reverting commits

```
git revert
```

Create a commit that cancels out previous commits

```
git revert <commit> # revert one specific commit
git revert HEAD~3.. # revert the last three commits
```

- `--no-commit` allows to create the reversing changes without commiting them immediatelly: Gives you more control

Exercise 1

1. Create a new directory (not in your current one, if it is tracked by Git!)
2. Change into it and initialize it for Git
3. Add a new text file “pet.txt” with the name of your favourite animal
4. Add this file to the staging area
5. Commit the change with a meaningful commit message

`mkdir`, `cd`, `git init`, `echo "dog" > pet.txt`, `git add --all`, `git commit -m "pet!"`

The .gitignore file

Ignoring files

By adding a (hidden) `.gitignore` file to your repository, you can tell Git to explicitly ignore certain files and directories. This is useful for

- `.log` files
- large datasets
- compiled/rendered/calculated output
- secrets

Make sure never to commit and push secrets (PWs, tokens, etc.) to a Git repository!

```
myFile.txt    # ignore a specific file
*.pdf         # ignore all PDFs (with a wildcard *)
logs/         # ignore a directory
logs/file.log # ignore a file in a directory
```

More patterns are available, see e.g. here

Unignoring files

Sometimes we want to ignore a pattern or directory, but not some specific subpatterns, directories or files within it

```
figures/           # ignore the figures directory
!figures/fig5.png # ... but track this one figure
*.pdf              # ignore all PDF files
!template/*.pdf    # ... but track all PDFs in this dir
```

Applying changes to the .gitignore file can be a bit brittle: Sometimes you have to empty the cache for a particular file affected by a change with `git rm --cached <file>`

Instead of unignoring a file in the .gitignore file, we can also add it manually with `git add -f <file>`

Branches

Creating a new branch

Branches allow you, to work off the main track. This is useful for

- Experiments and breaking changes
- Collaborative work: Suggesting changes
- Separate projects - e.g. GitHub pages (Dangerous!)

`git branch`

List and create branches

- `git branch <branch name>` creates a new branch
- Branches always "branch off" the current branch - like a tree
- New branches take current, untracked changes with them

Switching between branches

```
git switch <branch name>
```

Switch from one branch to the other

- `git switch` is only available since Git v2.23, users with older versions have to use `git checkout`
- Switching means, that Git will change the files in your directory: If a file exists only in one branch, then it will only be visible if you are on that branch
- Switching is only allowed, if changes on the current branch are properly commited

Merging branches

```
git merge
```

Integrate changes from one branch into another branch

```
      A---B---C otherBranch
     /
D---E---F---G master
```

```
git merge otherBranch # run on the master branch
```

```
      A---B---C otherBranch
     /         \
D---E---F---G---H master
```

- Only run `merge` when every change on both branches is committed
- When changes are contradictory, a **merge conflict** arises. We will talk about this case later

Exercise 2

1. Go back to your toy project from Exercise 1
2. Create a new branch named `eyeBranch`
3. Switch to this new branch
4. Edit `pet.txt` and add the number of eyes of this animal in another line (e.g. 2)
5. Create a commit for this change
6. Switch back to the master branch
7. Merge `eyeBranch` into the `master` branch

Beyond

Working with remotes

So far we have focussed on the a local directory, tracked by Git. But Git repositories can also be maintained on a remote server. That is the normal mode of operation for collaborative projects.

Git has multiple subcommands to interact with remotes:

```
git remote # List and modify the remote's URL
git clone # Download a local copy of a remote repository
git fetch # Download changes from the remote
git pull # Download and integrate changes from the remote
git push # Upload local changes to the remote
```

The details of these commands will be covered later

Tagging

`git tag`

Mark a specific point in the history of a repository with a name or number

- Often used for releases, but also valuable just as orientation points
- `-a` + `-m`: An annotated tag can be created with: `git tag -a v1.1 -m "Second submission after review"`
- `-l`: List tags
- `-d`: Delete a tag
- Tags can also be inspected: `git show v1.1`
- `git push` does not push tags, they have to be pushed explicitly with `git push origin v1.1`

Stashing

`git stash`

Put away changes to quickly go back to a clean working directory

A quick-and-dirty way to get to the HEAD without loosing intermediate work (unlike `git reset` and `git clean`)

```
git stash --all # Stash away the current work
git stash list # List the available stashes
git stash show # Show changes in the stash
git stash apply # Bring the changes in the stash back
```

- `show` and `apply` can also be applied only for specific stashes
- `git stash` has more subcommands that allow for very precise handling of stashes (not recommended)

Further obscure commands down the rabbit hole

Git has many more, very specific features, which are not required in daily life, but can feel like a superpower, when properly mastered

Some examples from rarely used to obscure:

```
git blame # Show which line in a file was edited by whom
git rebase # Moving a sequence of commits to another point
git range-diff # Show the difference between commit ranges
git cherrypick # Reapply the changes of an old commit
git notes # A system to attach meta-information to changes
git bisect # Search for the commit that introduced a change
           # Apparently brilliant for bug-hunting!
git worktree # Work with multiple worktrees
             # Multiple branches per repository?!
...
```