

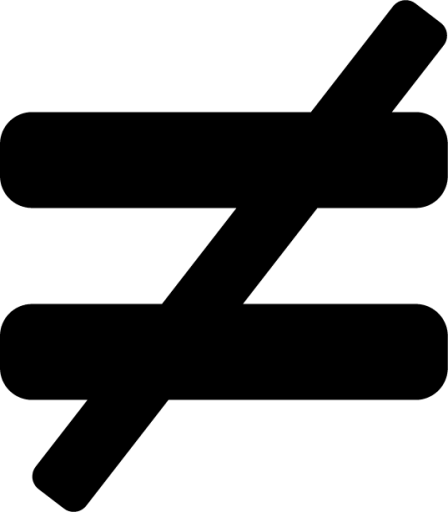
|  |  |
| --- | --- |
| **Common commands** | |
| **Command** | **Explanation** |
| pwd | Returns the active directory location. |
| ls | Returns a list of a files and directories. |
| cd directory name | Changes the current directory too the following directory. |

|  |  |
| --- | --- |
| **Making changes to a repo (project)** | |
| **Command/ keybinds** | **Explanation** |
| git add file\_name.extention | Adds a single file too the staging area. |
| git add . | Adds all files in the active directory too the staging area. |
| git commit -m "log message" file\_name.extention | Commits singular stage draft with log message. |
| git commit -m "log message" | Commits all the staged draft(s) with log message. |
| git status | Returns which file have changes in them that aren’t in the staging area yet AND returns which files are in the staging area and aren’t committed yet. |

|  |
| --- |
|  |

|  |  |
| --- | --- |
| **Git workflow** | |
| 1 | Modify a file. |
| 2 | Save the draft in the staging area. |
| 3 | When tried and test commit the update file. |

|  |  |
| --- | --- |
| **Editing file commands** | |
| **Command/ keybinds** | **Explanation** |
| **Using nano to edit a file.** | |
| nano file\_name.extention  (e.g. .csv, .txt) | Opens a text editor in which you can:  \* Delete.  \* Add to.  \* Make changes to a file. |
| *Ctrl + 0* | Save changes. |
| *Ctrl + X* | Exit text editor and return to shell. |
| **Using echo to create / edit a file** | |
| echo "text" > file\_name.extention | Creates a new file in the active directory containing the text between " ". |
| echo "text" >> file\_name.extention | Adds the text between "" to the designated file in the active directory. |



|  |  |
| --- | --- |
| **Find hidden folders** | |
| **Command** | **Explanation** |
| ls -a | Reveals hidden folders like the .git folder. |

|  |  |
| --- | --- |
| **Check the version of Git** | |
| **Command** | **Explanation** |
| git --version | Returns the version of git. |



|  |  |
| --- | --- |
| **Remote repos** | |
| Instead of having the git saved locally you can use platform like GitHub, Bitbucket or Gitlab to host remote repos.  Git stores the location of the original repo using a tag in the configuration. | |
| **Command/ keybinds** | **Explanation** |
| git clone <path-to-project-directory>  <optional name> | Clones (copies) a local repo located at the path and if specified gives the new repo a different name. |
| git clone <[URL]> | Clones a remote repo (e.g. form GitHub) |
| git remote add <name> <URL> | Creates an alias for the remote (making it easier to merge back to the remote repo). |
| git remote -v | If in a repo, git remote returns the list of the remotes (original repo name). If -v is added it also returns the URL/original location. |



|  |  |
| --- | --- |
| **Remote repos 3 push** | |
| Start by pulling again and then by merging and saving all changes locally. | |
| **Example** | **Explanation** |
| git push <remote alias> <local repo branch> | Pushes the local branch to the remote repo. |

|  |  |
| --- | --- |
| **Remote repos 2 pull** | |
| **Command** | **Explanation** |
| git fetch <remote alias> <local branch> | Fetches the latest remote version (generally main branch). |
| git merge <remote alias> <local repo> | Merges the fetched main branch with the local main branch to get it up to date. |
| git pull <remote alias> <local repo branch> | A combination of the two above commands in one. if –-no—edit is placed behind no message has to be given. |

|  |  |
| --- | --- |
| **Making a new repo (project)** | |
| **Command** | **Explanation** |
| git init <name> | Creates a new Git repo under the given name. (Git creates a new subdirectory (.git) in the directory you are at that moment located).  If you use git init while in a directory containing file Git automatically puts these files inside the repo. |





|  |  |
| --- | --- |
| **Retrieve settings** | |
| **Command** | **Explanation** |
| git config –list (--local/ --global/ --system) | Returns a list of customizable settings, additionally can add –local for settings for one specific project, --global for settings for all projects and –system for settings for all user on this computer. (user. email, name, core. editor, core. Reprository version, core.filemode, core. bare,core.logallrefupdates). |

|  |  |
| --- | --- |
| **Configure settings** | |
| **Command** | **Explanation** |
| git config –<level> <setting> <value> | Changes on the level e.g. global the indicates setting too the value. It needs to be between ' ' if it contains any spaces.  e.g. git config –global user.name 'robingoldenberg' |

|  |  |
| --- | --- |
| **Git log** | |
| **Command** | **Explanation** |
| git log | Returns an overview of all commits to a repo (project) in chronological order. |
| git log -3 | Returns the log of the number of most recent commits to a repo. |
| git log -4 file\_name.extention | Returns the log of only the number of most recent commits of the specified file. |
| git log –since='Apr 2 2022' | Returns the logs of commits made since the indicated date. |
| git log –since='Apr 2 2022' –until='Apr 11 2022' | Returns the logs of commits made between the indicated dates. |
| *space bar* | Move through the recent commits. |
| *q* | Exit the log. |
| git show hash[:8] | Returns the diff comparing that commit with its previous one  (hash being the unique identifier). |
| git show HEAD~1 | To look at changes made to files in a specific commit (ranging from the most recent HEAD to any before that using ~integer. |

|  |  |
| --- | --- |
| **Cleaning a project** | |
| **Command/ *keybinds*** | **Explanation** |
| git clean -n | Returns a list of files in the active directory that are currently not being tracked by GIT. |
| git clean -f | Delete all files from the active directory that aren’t tracked by GIT. |

|  |  |
| --- | --- |
| **Changes per document** | |
| **Command/ *keybinds*** | **Explanation** |
| git annotate file\_name.extention | Returns all changes to the specified file returning:  hash: first 8 digits of the commits unique id  author: who made the change  time: when was the change made  line: which line was changed  line content: part of the specified line. |





|  |  |
| --- | --- |
| **Ignore files** | |
| **Command/ *keybinds*** | **Explanation** |
| nano .gitignore | Creates a file called .gitignore after which you add the file names you want to ignore or \*format e.g. \*.log, \*.csv to ignore all files ending with that format. |

|  |  |
| --- | --- |
| **Branches** | |
| Branches avoids subdirectories and allows multiple users to work simultaneously without losing a working version. It keeps track of all changes. Each branch should be used for a specific task and then merged back into the main branch. | |
| **Command/ *keybinds*** | **Explanation** |
| git branch | Returns a list of all branches, the current branch your operating in is indicated by the \*. |
| git branch -m <name1> <name2> | Renames a branch (name1) to (name2) |
| git checkout <name> | Switch to the indicated branch. |
| git switch <name> | Switch to the indicated branch. |
| git switch -c <name> | Creates a new branch and switches to it. |
| git checkout -b <name> | Creates a new branch. |
| git diff <branch> <branch> | Returns the differences between 2 branches. |
| git merge <source> <destination> | Merges source branch into the destination branch. |
| git branch -d <name> | Deletes a branch. (only when it has been merged to main). |
| git branch -D <name> | Deletes a branch. (even when it hasn’t been merged to main). |
|  | |

|  |  |
| --- | --- |
| **Alias commands** | |
| **Command** | **Explanation** |
| git config –<level> alias.<alias name> '<command>' | Typically used to shorten a command for frequent use. E.g. git config –global alias.ci 'commit -m'. |
| git config--global --list | Returns a list of aliased commands and their original command. |





|  |  |
| --- | --- |
| **Branches 2 Merge conflicts** | |
| If during a merge Git returns an Automatic merge failed message it encountered conflicts that need to be fixed first. If this happens you have to open (nano) the conflicted file and change the conflict areas. After which you can stage, commit it to the main branch. After that you can try to merge the branches again. Best is to avoid merge conflicts by avoiding editing the same file in multiple branches. | |
| **Example** | **Explanation** |
|  | The Arrows too the left and the word HEAD indicates that lines beneath it contain the files contents in the latest commit of the current branch (destination).  The line of = signs refer to the center of the conflict as it is straight after the < signs it indicates the liens beneath are from the source version. If = sign is after content it indicates both files have different content on the same lines.  The > signs and the word update indicate that the source file has additional lines not found in the destination. |

|  |  |
| --- | --- |
| **Comparing files/branches** | |
| **Command/ *keybinds*** | **Explanation** |
| git diff --staged file\_name.extention | Compares the non-staged version with the committed version. |
| git diff -r HEAD file\_name.extention | Compares the most recent staged version and the committed version. |
| git diff -r HEAD | Compares all staged files with the last committed versions. |
| git diff -r HEAD~1 | Compares all the staged files with second to last(etc.) recent commit. |
| git diff hash[:8] hash[:8] | Compares 2 commits to each other selected by their hashes. |
| git diff HEAD~1 HEAD~2 | Compares 2 commits to each other selected by their commit position. |
| -r Indicating the selection of a specific version.  HEAD Indicating the most recent version.  HEAD~integer To indicate how many version beyond the most recent   version to compare too.  hash Being the unique identifier of the commit. | |
| **Returned results** | |
| @@ Indicates the location of the changes.  - red lines Are removed lines.  + green line Are added lines. | |



|  |  |
| --- | --- |
| **Un-staging** | |
| **Command/ *keybinds*** | **Explanation** |
| git reset HEAD file\_name.extention | Un-stages the staged version of the specified file. |
| git reset HEAD | Un-stages all files in the staging. |



|  |  |
| --- | --- |
| **Reverting a commit (all files)** | |
| **Command/ *keybinds*** | **Explanation** |
| git revert | Reverts to a previous version of the repo and makes a commit. (All files edited in the last commit are affected). |
| git revert <hash> | Reverts the commit relating to the hash. |
| git revert HEAD ~1 | Head Tilde 1 would undo the second -most recent commit. |
| git revert --no-edit head | Reverts the latest commit without committing at the end. (brining into the staging) |

|  |  |
| --- | --- |
| **Un doing changes (single file)** | |
| **Command/ *keybinds*** | **Explanation** |
| git checkout -- file\_name.extention | Undo all the changes that are made to a specified un-staged file. |
| git checkout . | Undo all changes that are made to all un-staged files. |
| git checkout hash[:8] file\_name.extention | Revert the specified file to a version from a specific commit. |
| git checkout HEAD~1 file\_name.extention | Revert the specified file to a specified historic version ( e.g. second to last). |
| git checkout hash[:8] | Restores all files in the project to their version in a specific commit. |
| git checkout HEAD~1 | Restores all files in the project to a specified historic version. |

|  |  |
| --- | --- |
| **Unstaging staged files** | |
| **Command/ *keybinds*** | **Explanation** |
| git restore --staged file\_name.extention | Removes a specified file from the staging area. |
| git restore --staged | Unstages all files in the staging area. |



|  |  |  |
| --- | --- | --- |
| **Merge types** | | |
| **Command** | **Merge type** | **Explanation** |
| Git merge branche --ff-only | Fast-forward merge | The default merge strategy.  Ideal for short-lived branches with simple changes and the main branch is unchanged.  Only merges the final commit of the feature branch. |
| Git merge -–no-ff branche | Recursive merge | Creates a merge commit with two parents preserving the branch its commits.  Ideal for long-lived branches and when main branch changes during the feature branch its lifetime. |
| #checkout the main branch  Git checkout main  #First in the branch  Git merge -–squash branch1  #Than commit to main git commit -m "commit text" | Squash merging | Combines all commits of a feature branch into a single new commit on the target branch (losing feature branch commit history).  Ideal for long-lived branches but where there is less of a need for the keeping of the commit history. |
| #Switch to main branch then:  Git merge -s octopus branch1 branch2 branch3 | Octopus merge | Used to merge three or merge branches at once in a single commit  Ideal when branches don’t have conflicts. |



|  |  |
| --- | --- |
| **Rebase** | |
| Rebase is used to rewrite history. Removes merge commits but retains a linear commit graph for clarity. Hereby letting you edit commits merging them and hereby cleaning up the commit history. Also, can be used to keep feature branches updated with main or clean them up before merging. | |
| **Command/ *keybinds*** | **Explanation** |
| Git rebase branch1 | Base way to rebase any conflicts need to be resolved manually |
| Git rebase -i <commit\_hash or eg HEAD~3> | Interactive rebase: Lets you edit multiple commits in the history at one time. |

|  |  |
| --- | --- |
| **GitHub Keywords** | |
| **Keyword** | **Explanation** |
| GitHub | Cloud-based hosting service used to upload and track its work. (Other example GitLab, bitbucket). |
| GitHub vs Git | GitHub uses Git but stores in cloud. |
| GitHub repo | A Remote repo of all the files related to a project and its history. |



|  |  |  |  |
| --- | --- | --- | --- |
| **Markdown syntax (short version)** | | | |
| **Syntax** | **Description** | **Syntax** | **Description** |
| # | Biggest heading (with a line under it) | ![<text>] (<url>) | Image if it fails to load the <text> is shown. |
| ###### | Smallest heading. | @<github username> | Tags a GitHub account. |
| \*\*<text>\*\* | Bold text. |  |  |
| \*<text>\* | Italic text. |  |  |
| [<text>] (<url>) | Hyperlink. The text between [] is clickable and sends you too the url. |  |  |

|  |  |
| --- | --- |
| **Repo Ui** | |
| **Page** | **Explanation** |
| Code | Here the repos files are visible, and the description of the project is visible and can be edited. Furthermore, from here you can create branches. |
| Issues | Here tasks and problems are tracked and communication with others will happen here. |
| Pull requests | Request to make a change to the project. (suggestion box) you can view them and accept them |
| Settings | Here you can make changes to the repo, including name and access permissions. |
|  |  |
|  |  |



|  |
| --- |
| **Setting up a repository** |
| There are two methods to set up a new repo, one is through the UI in the top right corner. |
|  |
| Or in the repositories view click the new button. |
|  |
| Once clicked you can import an existing repo or copy an existign repo template.  Name the repo and give it an description and set it to public or private.  You can select if a README and Gitignore file file should be created for the repo.  You can select the license = None default copyright laws apply. |



|  |
| --- |
| **README** |
| The readme its contents will always be visible at the bottom of the code page under all the files. The README file is a markdown file and can be edited from the Code page. |
| **Writing a README** |
| the README should contain list of contents of the repo, clearly explain the project to others.  Needs to have:  \*A title.  \*Table of contents.  \*How the project came about and the motivation about it.  \*Description of used technology, why \*Description of the process used.  \*Limitations.  \*Challenges that were encountered.  \*What problem the code hopes to solve.  \*What the intended use is.  \*Credits. |



|  |
| --- |
| **Edit/ Delete file** |
| Once a file is opened in GitHub on the top right you have a few icons pressing the pencil will let you edit the file, while pressing the trash can will let you delete the file. |
|  |
| To safe the file with the edits or the deletion of the file all you have to do is write a commit message and make a commit. |

|  |
| --- |
| **Creating new directory** |
| You can do this by clicking the add file button during the file creation process.  Empty directories are not allowed by GitHub. |
| After this if you use:  <directoryname> / <file name>  it will create a sub directory, durring the file creation process. |



|  |
| --- |
| **Adding a local file** |
| You can do this by clicking the add file button and upload files. |
|  |
| Than drag your local file into the repo and same as creating a new file write a commit message and commit. |

|  |
| --- |
| **Creating new file** |
| You can do this by clicking the add file button. |
|  |
| After this you can add the name of the file and extention. |
| After adding any content you can add an short commit message and optionally an extended description, before commiting it to the main branch. |

|  |  |
| --- | --- |
| **Creating a new branch** | |
| From the code page you can see which branches exists and by clicking the branch button next to the current branch and then the new branch button.  Enter a branch name and on which branch it should be based (branch source). Then press create branch.  To add rules to branches to protect the main branch by e.g. forcing a pull request before a commit, you have to add this in the settings under the branches tab. |  |





|  |
| --- |
| **Collaboration in a private repo** |
| To give access to a private repo to a colleague, you have to go to settings under collaborators. Here you can add people using username or email. |



|  |
| --- |
| **Forking** |
| Forking is the copy a repo without linking to the original repo. Anyone with a GitHub account can fork a public repo. |
|  |
| You can fork a repo by pressing the fork button. Pressing creates a new fork. |



|  |
| --- |
| **Cloning a repo** |
| To clone someone else's repo you move to the code tab, press code and copy the HTTPS. |
|  |
| In the Git terminal you can use "git clone <copied https>" you can than be asked to enter your Github account name and personal access token.  This cloned repo is than still linked to the original. |

|  |
| --- |
| **Personal access tokens** |
| A PAT is an alternative to a password for terminal commands. It is required for use of GIT(HUB). |
| **Creation** |
| Go to the settings page and then too developer settings. Then go to personal access tokens and press Generate new token.  After this you can give the token a name, an expiration time and scope of the pat. After which you press generate Pat, page will refresh, and the key will be displayed. |



|  |
| --- |
| **Create new issues** |
| To create a new issue, go to the issues tab. From there press the new issue button. |
|  |
| Give it a tile and add its details in a md format if an # is used you can link it to an existing issue. You can also assign it to people, give it a label or attach it to a milestone. After which you press the Comment button.  By pressing right mouse button on an existing issue you get the option to quote the comment in your own one. |

|  |
| --- |
| **Pull request** |
| A pull request is a way to notify others about a change you would like to make to a branch within a repo.  It allows the repo owner to check changes before they are added.  The intention of a pull request generally is to merge two branches. |
|  |
| To create a pull request, you go to the pull request tab.  After which you press the new pull request button. Than you chose the base branch (which is branch where we want to add our changes too), and the compare branch( which is the branch containing our changes).  After which you write some information about the intended changes, through a title and description, and assign someone too review the pull request. |



|  |
| --- |
| **GitHub projects** |
| GitHub projects, is a project hub hosted by GitHub. It can be used for sprints and organizing tasks and tracking their progress.  It integrates with issues and pull requests within GitHub |
|  |

|  |
| --- |
| **Review a pull request** |
| To review someone else's pull requests, you go to the pull request tab and click the pull request to review. |
|  |
| After which move too the Files changed tab. Where you can review the proposed changes.  you can add command to each line by pressing the + sign. After which you can approve it, just comment without giving approval or requests changes to be made before merging (denying the request).  When you have approved you can click merge pull request and confirm the merging of the branches.  After which the old branch can be restored or deleted. |



|  |
| --- |
| **Setting up GitHub project 1** |
|  |
| Within a GitHub repository go the Projects tab and press “new project” |
| Choose a name. Choose a template (can customize afterwards):  Table = just a spreadsheet where you can add tasks and their statuses. Board = a sprint board.  Roadmap = it’s a board but also contains a timeline.  Ect. |
| Add cards representing tasks or issues. |



|  |
| --- |
| **Setting up GitHub project 2** |
| You can configure both visibility and access to the project. |
| **Visibility settings** |
| Public: visible to everyone.  Private: restricted to team and collaborators. |
| **Access Roles:** |
| Admin: Ful control.  Write: Can modify.  Read: View only  No Access: Cant even see the board |



|  |
| --- |
| **Setting up Insights** |
| To create a chart, click ”insights” and then ”New chart” |
| Click ”configure” to customize the x-axis and group by status. |

|  |
| --- |
| **Setting up GitHub actions** |
| Open the project and click the three dotted menu in the top right corner and choose workflows. |
|  |
| GitHub provides a list of default workflows e.g. automatically moving tasks to done when an issue is closed, |
| To customize a workflow select ”Edit” in the top right corner |



|  |  |
| --- | --- |
| **GitHub projects tools** | |
| There are two key tools in GitHub projects | |
| **GitHub Actions** | **Insights** |
| Used for automating workflows, simplifying processes. | Used for tracking progress, spotting issues, making informed decisions. |
| It can automatically move task based on events, keep boards updated automatically, enables to create custom workflows, tagging issues based on criteria. | Helps to visualize progress, identify bottlenecks using charts. |



|  |  |
| --- | --- |
| **GitHub advanced Security** | |
| GitHub advanced security provides essential tools for securing code. | |
| Code scanning | Identify vulnerabilities in side the code. |
| Secret scanning | Detect and protect sensitive data. |
| Dependency Graph & Dependabot | Manage, visualise dependencies and automate security updates. Found under the insights tab. |



|  |  |
| --- | --- |
| **GitHub organization Authentication methods** | |
| Two factor authentication | Adds extra security level. |
| Secured socket shell keys | Secure key-based access. |
| Personal access tokens | Controlled API access. |
| Identity providers | Centralized user management. |

|  |
| --- |
| **GitHub Organization** |
| A GitHub Organization is a centralized workspace to manage multiple projects, with role-based access and security policies. |
|  |
| To create a new organization click on profile and select your organizations. Than select new organization and choose a payment plan, fill in the needed info. |

|  |  |
| --- | --- |
| **GitHub organization roles** | |
| Owner | Full control of the organization. Can create teams. |
| Members | Standard access for collaboration. Can create teams. |
| Outsiders | Limited repo access. |
| Moderators | Manager interactions in public repos. |
| Billing managers | Handles billing settings. |
| Security managers | Oversees security settings. |
| **Team Roles** | |
| Member | Collaborates on projects. |
| Maintainer | Manages settings and members of the team. |



|  |
| --- |
| **Setting up GitHub actions 2** |
| Github actions makes it possible to automate tasks directly in the repository using YAML files. |
|  |
| These workflows are triggered by events like code pushes. |
| Common tasks: schedule data pipelines, setting up environments, deploying machine learning models to production. |



|  |  |
| --- | --- |
| **Remove sensitive Data from a Repo** | |
| When finding accidentally pushed sensitive data there are two tools that can be used to delete the info from both the repo and the repo history. | |
| **Git filter-repo** | **BFG repo-cleaner** |
| Allows for detailed management of history. | Faster and simpler to use. |
| Best for intricate cases. | Great for large-scale deletions. |
| Supports various tasks. | Less control and not as fast. |



|  |
| --- |
| **Setting up security policies** |
| **Setting up a Security.md file** |
| The file guid users how to report security issues and outlines actions. E.g. reporting guidelines and response times. |
| Setting up the security.md file can be done from the Security tab, then select security policy and start setup. |
| After setup it can be found back in the security tab. |