# Version Control of Code and Data

**Track, organize and share your work: An introduction to Git for research**

Lennart Wittkuhn Konrad Pagenstedt

# Contents

# Preface


Figure 1: This illustration is created by Scriberia with The Turing Way community. Used under a CC-BY 4.0 licence. DOI: 10.5281/zenodo.3695300 (Version 3, direct download link).

Welcome to the world of version control! The purpose of this book is to empower scientists, researchers, and students with the knowledge and skills needed to use Git for version control of code and data.

## Who is this book for?

This guide is meant to be a gentle introduction to version control for (aspiring) scientists, who are seeking to become more effective in managing the evolution of digital objects on their computers. Whether you're conducting experiments, writing code, collaborating with scientific peers, or managing complex data sets, Git provides a robust framework to enhance the efficiency, reproducibility, and collaboration of your work. While this book was developed with scientists in mind, it's of course open to anyone who wants to learn more about Git. We try to avoid technical jargon as much as we can. When we discuss best practices in using Git commands, we try to offer multiple alternatives but also give opinionated recommendations as guidance to new users.

## What is the purpose of this book?

Version control can be a real game-changer for your scientific projects. By adopting Git, you gain the ability to trace the evolution of your work, experiment with new ideas without fear of irreversible consequences, and collaborate with a global community of researchers. Of course using Git also

adds some layer of complexity to your workflow, especially in the beginning. We aim to simplify and demystify this versatile tool for you. Whether you're new to version control or have dabbled in it before, this book aims to add something for all levels of expertise.

## How to use this book?

Git is fundamentally a command-line tool which means you typically interact with it by typing text-based commands into a small command-line window rather than clicking on buttons in a graphical user interface (GUI) as in many other applications. This book focuses on teaching Git from the command line. While the command line is arguably the rockier road to learning Git, we believe that it provides more long-term benefits and allows to make use of the full potential of Git. This book therefore covers basics of the command line to teach you just enough to interact with Git via the command line effectively. That being said, if you prefer to interact with Git via a graphical user interface, you can still learn about the fundamental concepts of Git in this book and then implement them in your preferred GUI.

We also believe in learning by doing and try to focus on implementation as much as possible. The concepts introduced in this book are accompanied by practical examples, hands-on exercises and quizzes. Feel free to follow the exercises to gain the necessary "muscle-memory" to start using Git in your day-to-day work. Try out the commands in each chapter, play around with them or apply them to a project of yours!

This book was initially created for the full-semester course "Effective Progress Tracking and Collaboration: An Introduction to Version Control of Code and Data" at University of Hamburg, Germany. The book is therefore structured for a course that is spread out across multiple sessions. Each chapter and the accompanying quizzes and exercises should roughly fill 90 minutes of class time during a course.

## How can I contribute?

This book is constantly evolving and meant as a living resource, and your input can make it even better! If you spot typos, have suggestions for improvement, or want to contribute new content, we welcome your involvement! If you find a typo, an unclear explanation, have an idea for a new chapter or want to see a specific topic covered you are more than welcome to open an issue or submit a pull request in the GitHub repository of this book.

## Testimonials

Testimonials of students enrolled in the course "An Introduction to Version Control of Code and Data" at University of Hamburg, where this book acted as the primary learning resource:

> The online, tailor-made course book and wiki, along with the exercises and quizzes were incredibly helpful and aided my comprehension.

All materials are conveniently accessible through a course website and completely open source, thus conveying important principles of good scientific practice beyond version control. I particularly appreciated this open approach!

# Contents

The estimate of the **reading time** for each chapter is computed by counting the words in the chapter and assuming a reading speed of 200 words per minute.

# Learning Objectives

## Introduction to version control

- ☐ What is version control?
- ☐ Why is version control useful (for research)?
- ☐ What are Git and GitHub?
- ☐ What is the difference between Git and GitHub?

## Command Line

- ☐ Explain when and why command-line interfaces should be used instead of graphical interfaces.
- ☐ Understand how to form file paths and navigate directories.
- ☐ Understand the difference between absolute and relative paths.
- ☐ Understand how arguments and flags are used to modify command-line commands
- ☐ Understand the concept of wild cards (*)

## Installation

- ☐ Downloading and installing Git

## Setup

- ☐ You know how to set up Git for the first time
- ☐ You have set up Git on your computer
- ☐ You understand the different Git configuration levels
- ☐ You know how to configure your username and email address in Git
- ☐ You have set up your preferred text editor when working with Git
- ☐ You can escape the command-line text editor Vim

*Learning Objectives*

## First steps with Git

- ☐ Initializing a Git repository
- ☐ Staging and committing changes
- ☐ Exploring the commit history
- ☐ Comparing versions
- ☐ Creating a `.gitignore` file

## Branches

- ☐ Knowing purpose and benefits of using branches in Git
- ☐ Creating and switching between branches
- ☐ Merging branches and resolving conflicts

## GitHub

- ☐ Connecting Git and Github
- ☐ Setting up a GitHub repository
- ☐ Pulling and pushing changes to / from a Github repository

## Tags and Releases

- ☐ Understand why Git tags matter in version control and project management.
- ☐ Learn when to use lightweight or annotated tags and how to apply them.
- ☐ Get the hang of pushing and pulling tags in Git for seamless teamwork.
- ☐ Explore how GitHub releases complement Git tags
- ☐ Discover how to use Zenodo to make your repository citable

## Project Management

### Issues

- ☐ Understand the purpose of GitHub Issues
- ☐ Master the creation and management of Issues
- ☐ Collaborative problem solving with Issues
- ☐ Practicing with a practical exercise

## Graphical User Interfaces

- ☐ Understanding the benefits of Git GUIs
- ☐ Exploring different GUIs
- ☐ Exploring branch management in a GUI
- ☐ Practicing with a practical exercise

## Stashing and Co.

- ☐ Stashing and retrieving changes
- ☐ Undoing changes and removing files
- ☐ Rebasing and cherry-picking

# 1 Introduction

Introduction to fundamental concepts of version control.

Take the quiz! UHH WS 23/24

Learning Objectives
- ☐ What is version control?
- ☐ Why is version control useful (for research)?
- ☐ What are Git and GitHub?
- ☐ What is the difference between Git and GitHub?

## 1.1 What is version control?

> "Version control is an approach to **record changes** made in a file or set of files **over time** so that you and your collaborators can track their history, review any changes, and revert or go back to earlier versions." – The Turing Way Community [17], chapter on Version Control.

## 1.2 Benefits of version control

Version control offers numerous benefits. It allows to …

- Track the history of changes in your files (it's clear what you did & you have the option to revert back)
- Work with the latest version of a file
- Backup previous versions of files
- Go back to previous versions of your files
- Test out new features without messing up your previous version
- Collaborate with others on the same files at the same time

This makes version control a great data management and documentation tool.


Figure 1.1: This illustration is created by Scriberia with The Turing Way community. Used under a CC-BY 4.0 licence. DOI: 10.5281/zenodo.3695300 (Version 3, direct download link).

## 1.3 Challenges

### 1.3.1 Example

You have a person 1 who is developing a script to analyze their data. You go through different versions, script-v1, script-v2, script-v3. Now you want to test out a new feature. It takes you a while to test this feature. It becomes unclear what you should call all these files. Then you realize that you need help with this. So then researcher 2 comes in and they help you implementing this feature. More files are created, but no ideas how to name them. Person 3 has a lot of ideas but is not editing the files directly. So script v-3 is created. One you realize that this is becoming complicated, you create a meta document that you want to use to keep track of the project progress. Then you realize that you software has bugs and you can't figure out when this bug has occurred for the first time. Time has passed and you ask yourself: what is the latest version of my script and where is it? When did I incorporate that new feature. who made that change and why? Your collaborator asks you to run version 2 of the script but you dont have it anymore. In a real-life situation, there might be many more people involved and many more documents.

## 1.4 Versions in multiple files

If you are a student or researcher you might be familiar with the situation illustrated in Figure 1.2: You are writing a paper and you are really excited because you finally have a full draft. To mark this, you call the file `FINAL.docx` and send it off to your advisor. Your advisor has a few comments, so you integrate those changes into a new version that you call `FINAL_rev2.doc`. You show your draft to your advisor again and again they have even more comments. So as there are more and more rounds of feedback your file naming system an get out of hand very easily.

Apart from the issues with file naming, a deeper point of this illustration with respect to research is, that there is rarely a point in scientific work where we are really confident that something is indeed the "final" **TM** version.

If you can relate to the poor student Figure 1.2, it is quite likely that you have also already engaged in this form of version control.

More generally, this common way of implementing file versioning of files works by **appending versions or descriptive labels to the filenames**, or by **adding initials** at the end of the filename:

- `file_v1.docx`, `file_v2.docx`, `file_v3.docx`, etc.
- `draft.docx` to `draft_comments_LW_final_edited.docx`, etc.

Every time you make a critical change to a file, you duplicate it, rename it according to your versioning scheme and continue working in the duplicated file. While this approach might be manageable for a single file or user, it likely becomes messy when dealing with numerous files, repeated revisions, and multiple users, especially for large, long-term projects, as illustrated in Figure 1.2, Figure 1.3 and Figure 1.5. In such cases, more advanced version control systems may be necessary.

> Your number one collaborator is yourself from six months ago and she doesn't answer emails.

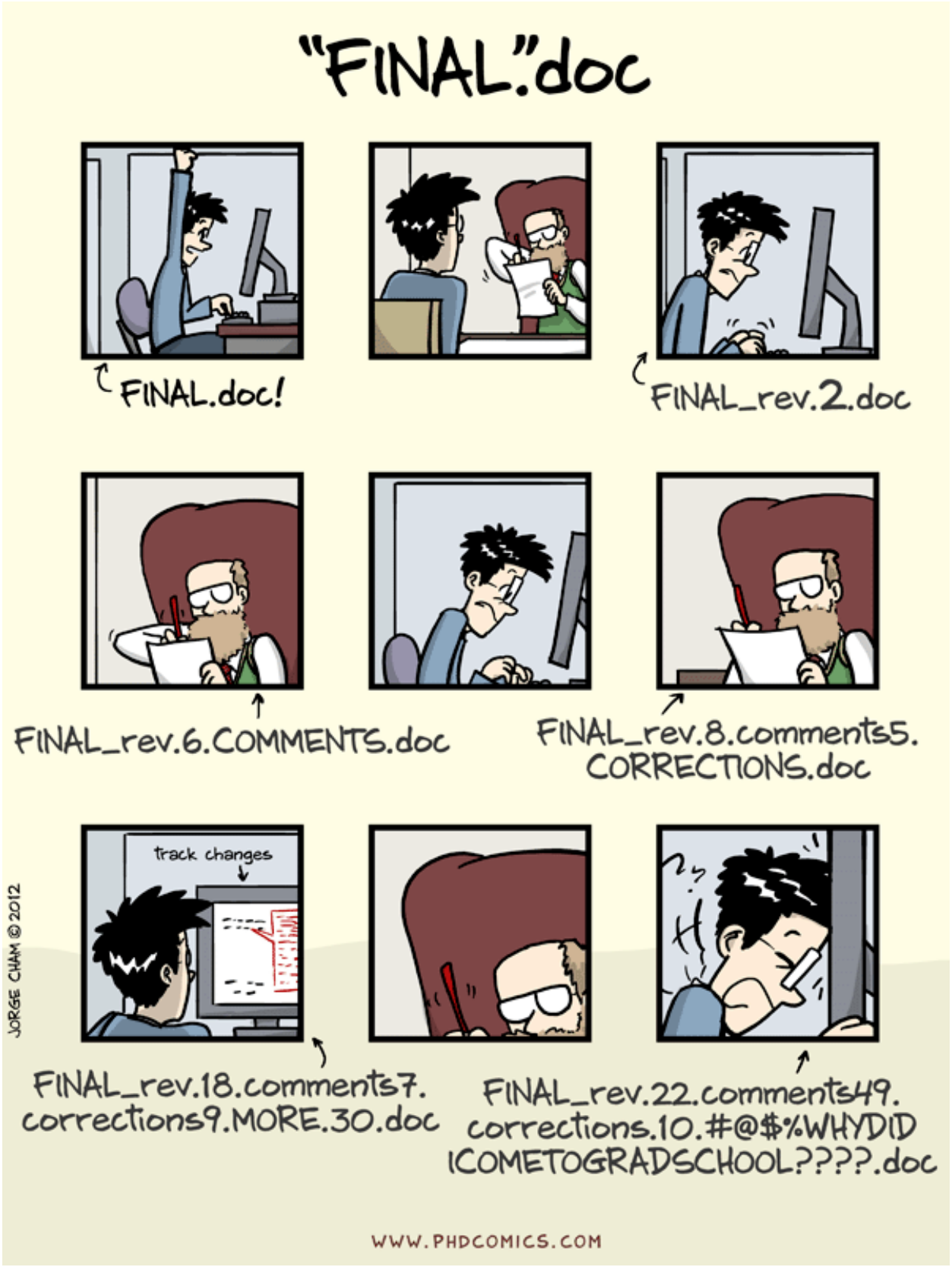



Figure 1.2: “notFinal.doc” - originally published 10/12/2012. “Piled Higher and Deeper” by Jorge Cham (phdcomics.com). All content copyright 1997-2023 Piled Higher and Deeper Publishing, LLC. (direct link).

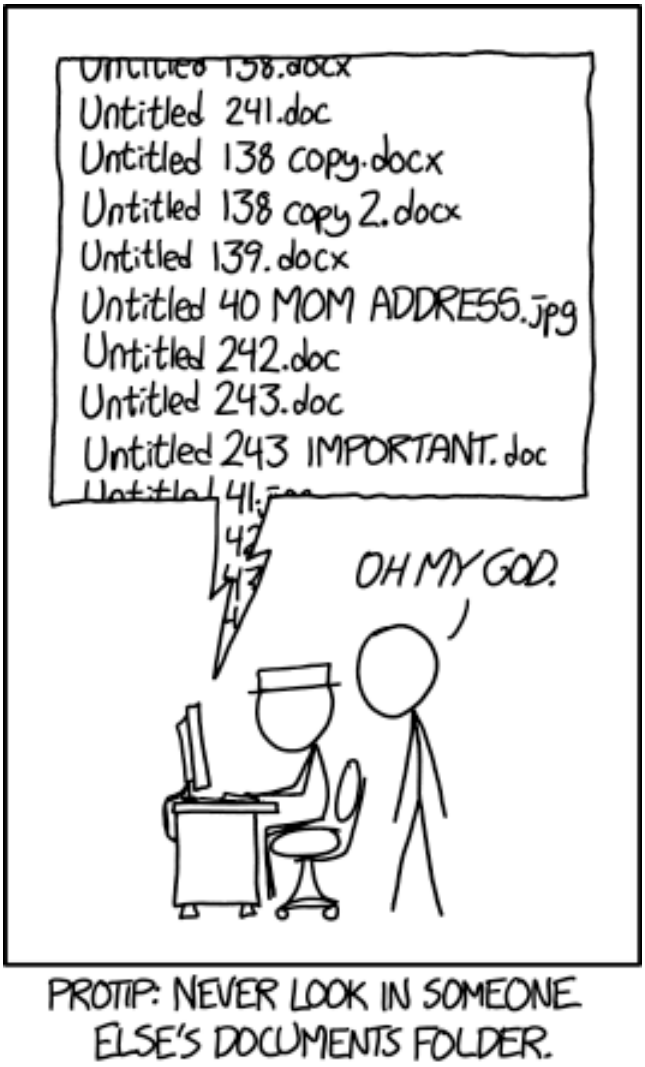



Figure 1.3: "Documents" by xkcd (xkcd.com). Used under a Creative Commons Attribution-NonCommercial 2.5 (CC BY-NC 2.5) licence (direct link).

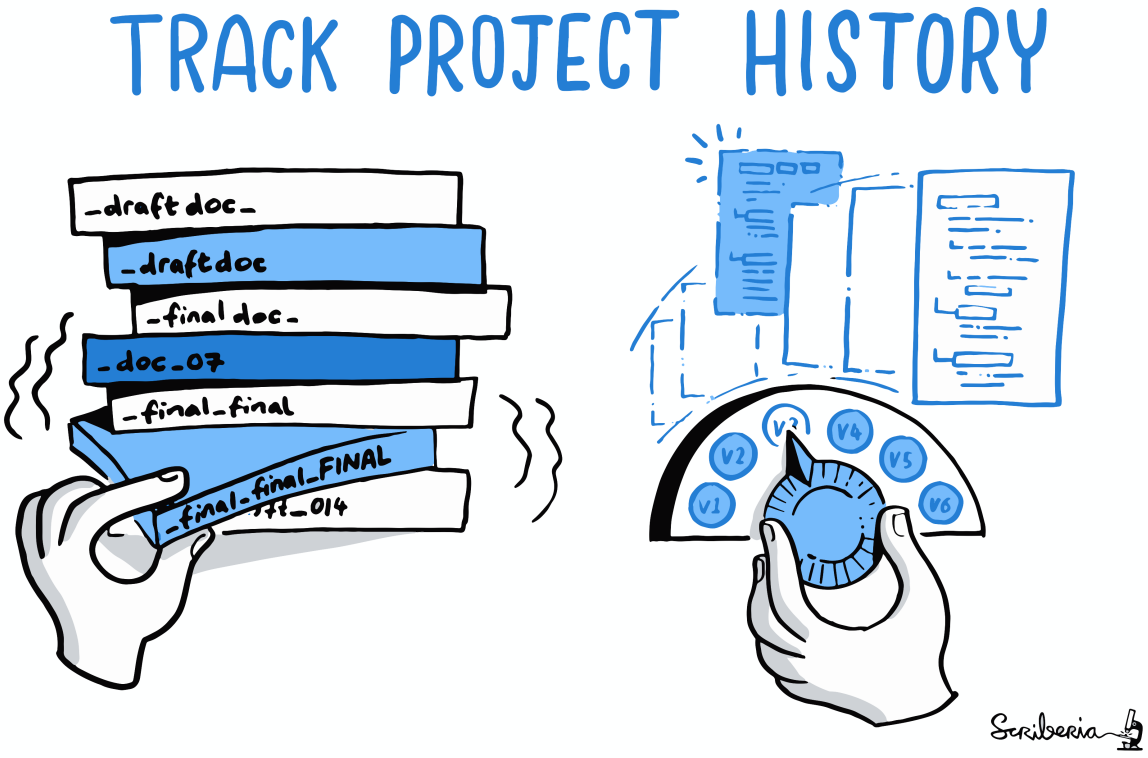



Figure 1.4: This illustration is created by Scriberia with The Turing Way community. Used under a CC BY 4.0 licence. DOI: 10.5281/zenodo.3695300 (Version 3, direct download link).

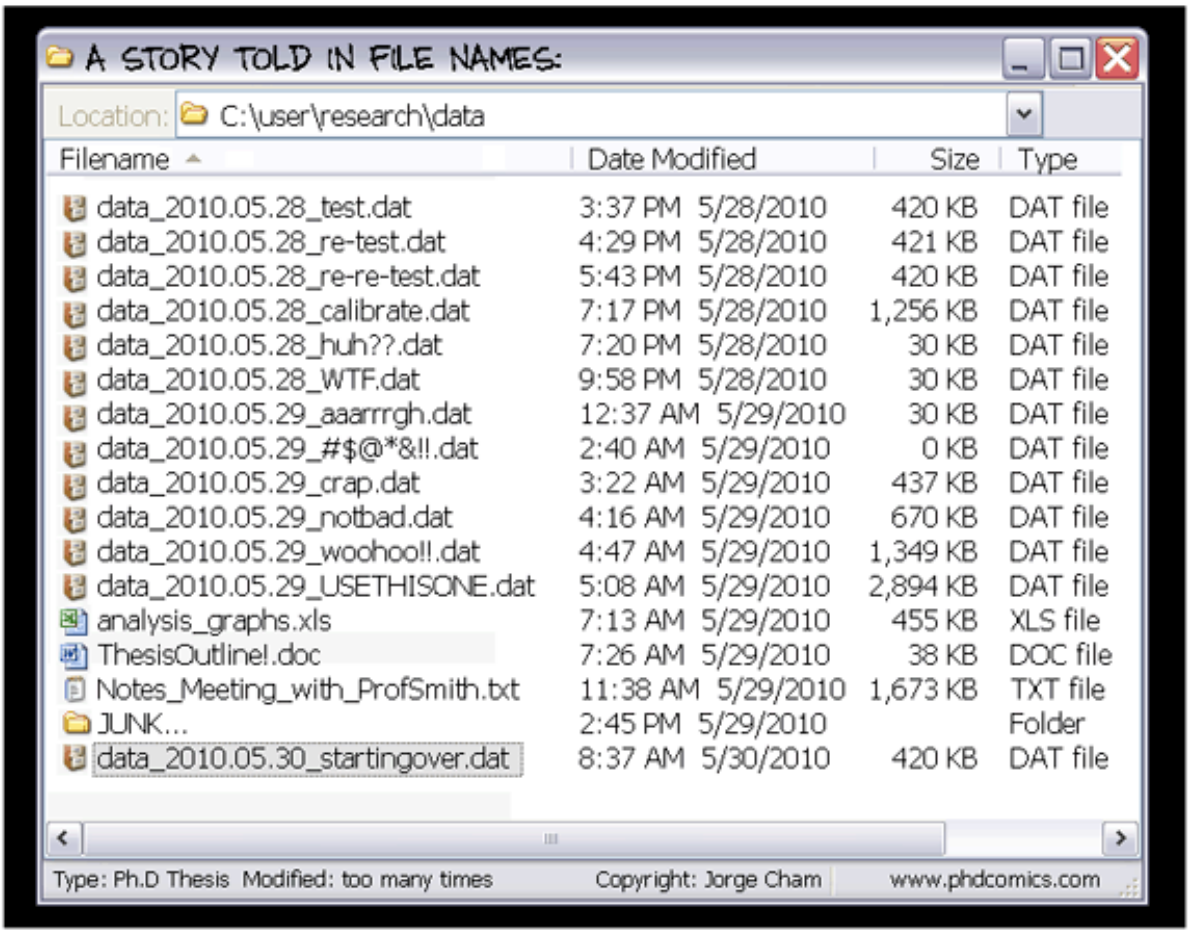



Figure 1.5: “A story told in file names”, originally published 5/28/2010. “Piled Higher and Deeper” by Jorge Cham (phdcomics.com). All content copyright 1997-2023 Piled Higher and Deeper Publishing, LLC. (direct link).

If you mostly work on your own or with a very small team, you might think that you will not encounter such situations. However, remember that your number one collaborator is yourself from six months ago and you will often find yourself in situations trying to remember what happened six months ago.

Imagine you were on a very long vacation (who would not like to imagine that!) and you would come back to a project folder as illustrated in Figure 1.5. Would you know which file to continue working with? Would you know how the different files related to each other? How could someone else start working with this project or how would you start working with this if you would get such a project folder from someone else? It is very difficult to understand how to work with this project and you will likely spend a lot of time figuring out what all the different files are, what they mean and why they are titled in such a way.

### 1.4.1 Versions in a single file

Some software can keep track of all the changes made to a file without making new copies of the file. This includes word processors like Microsoft Word's Track Changes, Google Docs' version history, or LibreOffice's Recording and Displaying Changes.

Disadvantages of version control using Word processors compared to Git

1. **Limited Collaboration:** Word processors' track changes feature is primarily designed for collaboration within the application, making it difficult to collaborate with users who don't have access to the same word processing software. In contrast, Git allows seamless collaboration across different platforms and tools.
2. **Limited Review Features:** Beyond comments in the text itself, word processors lack the advanced review features offered by Git, such as pull requests and code comments. These features facilitate the review processes, making it easier to provide feedback and discuss changes.
3. **Non-Atomic Commits:** In most word processors, tracking changes may lead to non-atomic commits, where multiple unrelated changes are combined into one. This can make it harder to understand the history and roll back specific changes when needed, compared to the precise commit system in Git.
4. **Limited Branching and Merging:** Git provides powerful branching and merging capabilities, allowing for the creation of multiple branches for different features or experiments. Word processors often lack this level of branching and merging, making it harder to manage complex development workflows.
5. **No Version Tagging:** Git allows tagging specific versions, marking them as milestones or releases. Word processors generally lack this feature, which can be essential for identifying important points in the development process.
6. **Lack of File-Level Versioning:** Word processors typically track changes at the document level, lacking the ability to track changes at the file level. Git, on the other hand, allows version control for individual files, enabling more granular control over the project's components.
7. **Limited Automation:** Word processors may offer basic automation features, but they can't match the automation capabilities provided by Git and GitHub, such as pre-commit hooks, continuous integration, and automated testing.

8. **Difficulty in Managing Large Projects:** Word processors' track changes feature may become less efficient and manageable as the project grows in size and complexity. Git, as a dedicated version control system, is better suited for handling large-scale projects.
9. **No Remote Repositories:** Word processors do not have built-in support for remote repositories like Git does. Remote repositories in Git enable decentralized collaboration and backup, making it easier to work with distributed teams.
10. **Dependency on Proprietary Software:** Using word processors for tracking changes ties the version control process to specific software vendors, limiting the flexibility and interoperability that Git offers as an open-source, widely-used tool.

This allows you to look at the file's history and see how it has changed over time. Examples of such software include Google Drive, Dropbox, and Overleaf. You may have noticed that these services automatically save different versions of your files and allow you to go back to previous versions if needed.

Disadvantages of Version Control using Dropbox and Google Drive compared to Git

1. **Lack of Fine-Grained Versioning:** Both Dropbox and Google Drive offer basic versioning features, but they often lack the granularity provided by dedicated version control systems like Git. In version control systems, you can track changes at the file level, while Dropbox and Google Drive may only keep a limited history of entire documents.
2. **No Code Collaboration Features:** While Dropbox and Google Drive allow file sharing and real-time collaboration, they do not provide specialized features for collaboration, like pull requests, or conflict resolution, which are often essential in the development of research outputs.
3. **Limited Branching and Merging:** Dropbox and Google Drive do not support branching and merging, making it challenging to manage complex development workflows, especially in collaborative projects.
4. **Dependency on Third-Party Servers:** When using Dropbox and Google Drive, version control relies on their servers and infrastructure. This dependence may raise concerns about data security, privacy, and the availability of the service.
5. **File Size and Storage Limits:** Both services have limitations on file sizes and storage space. This can be problematic for larger projects with numerous files or when working with large binary files.
6. **Limited Automation and Integration:** Dropbox and Google Drive lack the extensive automation and integration options provided by Git. In Git, you can set up various automated workflows and integrate with continuous integration tools for smoother development processes.
7. **Revision Conflicts:** Version control in Dropbox and Google Drive may not handle revision conflicts as effectively as Git. Resolving conflicts manually can be time-consuming and may lead to data loss or discrepancies.
8. **Limited Offline Capabilities:** When working offline, accessing version history or performing version control tasks in Dropbox and Google Drive might not be as seamless as with Git, which is designed to work effectively in both online and offline environments.

Figure 1.6 illustrates the concept of **tracking versions within a single file**. It illustrates a simplified linear project timeline (gray line). A version control system keeps track of the changes made to the file at specific time points (gray nodes in Figure 1.6). These nodes represent various actions such

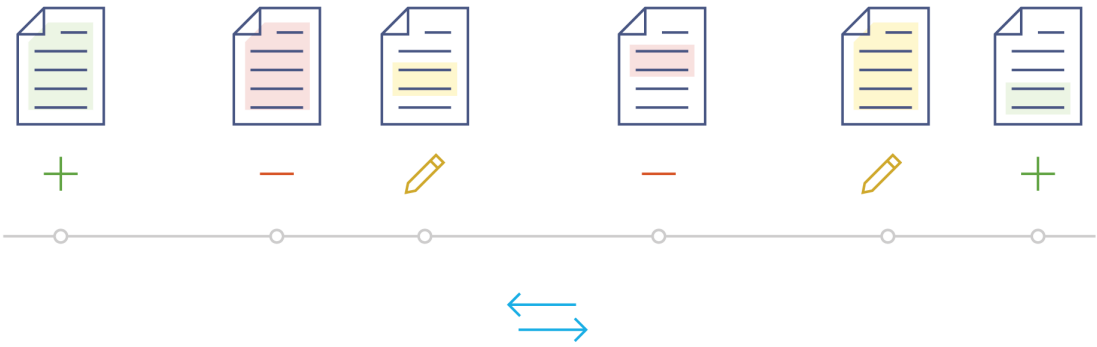


Figure 1.6: Image from Chapter 3.1 on “Collaboration, Version Control, & GitHub” of the Mozilla Science Lab’s Study Group Orientation handbook, used under a Mozilla Public License Version 2.0 (direct download link)

as additions (first node), deletions (second node), or edits to the content (third node). The blue arrows at the bottom indicate the ability to navigate along this linear timeline. By recording these changes, it becomes possible to access previous versions of the file. This means that you can go back and examine earlier versions or even revert the entire workflow back to a specific version, if needed. Rather than saving new files for each change, this system saves the changes within the same file, allowing for easier management and tracking of version history.

### 1.4.2 Features of a version control system

1. **What:** It keeps track of the changes made to a file when you edit it.
2. **When:** It records the date and time when each change was made.
3. **Who** It keeps a record of who made each change.
4. **Why:** It allows you to add a note explaining why you made a particular change.

The system takes snapshots of your project, giving them unique codes that include all these features. Instead of saving new files for each change, only the specific modifications made to the file are saved. This makes it easy to compare different versions, restore previous versions, and combine changes when needed.

## 1.5 Git

Among various version control systems, Git is the most widely used. It was created by the Linux development community in 2005. Git is a software primarily written in C, but you don’t need to know C.

Figure 1.7: Git Logo (full color Git logo for light backgrounds) by Jason Long is licensed under the Creative Commons Attribution 3.0 Unported License (link to download page).

### 1.5.1 Key features of Git

- Free, fast, and open source.
- It is a **distributed** system, meaning everyone working on a project has a complete copy of it.
- Most operations in Git can be performed using local files and resources, **minimizing the need for external dependencies**.
- Git keeps a **detailed history of changes**, including what was changed, when it was changed, who made the change, and why it was made.
- You can easily retrieve previous versions of your project, allowing you to see the progression of your work.
- Git forms the **foundation for collaboration tools** like GitHub and GitLab.

When you use Git, it takes snapshots of your project whenever you choose. It then compares the current snapshot with the previous one and prompts you to provide a message explaining why you made the specific changes.

Why is Git called Git?

The name “git” was given by the main developer Linus Torvalds when he created the software. Torvalds is well-known for his involvement in the development of the Linux kernel.

> *“I’m an egotistical bastard, and I name all my projects after myself. First ‘Linux’, now ‘git’.”* – Wikipedia: Git

For more insights and discussions about the name, you can refer to the README.md file on GitHub.
The name “git” was given by Linus Torvalds when he wrote the very first version. He described the tool as “the stupid content tracker” and the name as (depending on your mood):

- random three-letter combination that is pronounceable, and not actually used by any

common UNIX command. The fact that it is a mispronunciation of “get” may or may not be relevant.
- stupid. contemptible and despicable. simple. Take your pick from the dictionary of slang.
- “global information tracker”: you’re in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- “goddamn idiotic truckload of sh*t”: when it breaks

## 1.6 Summary

Git is a free **version control system** that helps you manage and track the history of your files, allowing you to retrieve previous changes. It is used **locally** on your computer.

GitHub is a popular **cloud-based** hosting service for sharing projects tracked with Git. It offers **collaboration** features and serves as a reliable backup solution.

## 1.7 How do I interact with git on my computer?

There are many options to interact with Git.

### 1.7.1 Command line / terminal

See chapter on command line.

### 1.7.2 Extensions to existing software

Your text editor or analytical software is very likely to have a Git extensions / plugin, often via a Graphical User Interface (GUI).

- RStudio Desktop
- Visual Studio Code

### 1.7.3 Git Clients (applications with a GUI)

Several software applications enable the use of Git on your local computer without the need for the command line. The following is a non-comprehensive list of such software. Please consult their respective documentation for further details.

- GitKraken
- GitHub Desktop

## 1.8 Common misconceptions

### 1.8.1 Misconception: Git is the same as GitHub

One of the most widespread misconceptions is that Git and GitHub are the same thing. Git and GitHub are two distinct but related tools in the world of version control.

**Git is a version control system** (software installed on your computer) that allows you to track changes and manage the history of your files locally on your computer. It provides features such as branching, merging, and reverting to previous versions.

**GitHub is a web-based hosting service** for your Git repositories, allowing you to store and share your Git projects online. GitHub provides additional collaboration features such as issue tracking, pull requests, and project management tools.

In summary, Git is the version control system itself, while GitHub is a hosting service built on top of Git. Git can be used independently without GitHub, but GitHub relies on Git for its functionality.

### 1.8.2 Misconception: Git is only for code and programmers

While version control systems like Git are often associated with software development (and are indeed highly valuable for managing code-related projects), they are not limited to code and not exclusive to developers. Version control can be beneficial for anyone working with files that undergo changes over time, including writers, designers, and data analysts.

### 1.8.3 Misconception: Git is only for collaboration

Git offers remarkable benefits for collaboration, but its value extends beyond working with others. It enables you to collaborate effectively with your past and future self, fostering valuable skills and boosting your confidence when collaborating with others. Even if you currently don't have a specific reason to collaborate with others, trying Git on your own can be highly beneficial and rewarding.

### 1.8.4 Misconception: Git auto-saves your changes

Unlike automatic saving, where changes are continuously saved in the background, Git requires manual intervention to capture snapshots. You have to explicitly inform Git when you want to take a snapshot, indicating that you want it to acknowledge your changes. While this may sound annoying at first, the benefits of having full control over your version history will become apparent in later chapters of this book.

## 1.9 Acknowledgements

We would like to express our gratitude to the following resources, which have been essential in shaping this chapter. We recommend these references for further reading:

| Authors | Title | Website | License | Source |
|---|---|---|---|---|
| Koziar et al. [10] | swcarpentry/git-novice: Software Carpentry: Version Control with Git 2023-05 | | CC BY 4.0 | |
| Bryan [4] | Excuse Me, Do You Have a Moment to Talk About Version Control? | | Website: https://doi.org/10.1080/00031305.2017.1399928. | |
| Ram [16] | Git can facilitate greater reproducibility and increased transparency in science | | CC BY 2.0 | |
| Perez-Riverol et al. [15] | Ten Simple Rules for Taking Advantage of Git and GitHub | | CC BY 4.0 | |
| AI for Multiple Long-term Conditions Research Support Facility [1] | Introduction to version control with git | | CC BY 4.0 | |

# 2 Command Line

In this session, we will explore the concept of the command line and discover some commands that work well alongside Git.

Take the quiz! UHH WS 23/24

Learning Objectives
- ☐ Explain when and why command-line interfaces should be used instead of graphical interfaces.
- ☐ Understand how to form file paths and navigate directories.
- ☐ Understand the difference between absolute and relative paths.
- ☐ Understand how arguments and flags are used to modify command-line commands
- ☐ Understand the concept of wild cards (*)

## 2.1 Why using Git from the command line?

Git is fundamentally a command-line tool. This is why comfort with the command line is essential for learning Git.

## 2.2 General advantages of the command line

1. **Longevity:** The command line has stood the test of time, remaining relevant and widely used even as new technologies emerge.
2. **Power:** The command line is a robust and powerful tool, enabling users to accomplish intricate tasks efficiently.
3. **Simplicity:** With just a few keystrokes or lines of code, the command line empowers users to execute complex tasks effortlessly.
4. **Task Automation:** The shell allows users to automate repetitive tasks, saving time and effort in performing them manually.
5. **Workflow Integration:** Users can seamlessly combine smaller tasks into larger, more potent workflows, enhancing productivity and efficiency.
6. **Comprehensive Feature Set:** The command line interface provides access to the complete range of Git features, offering more extensive functionality compared to graphical user interfaces (GUIs).

7. **Extensive Online Support:** When seeking help or troubleshooting Git-related issues online, you will often find valuable assistance in the form of command line instructions. Platforms like Stack Overflow frequently provide guidance and solutions based on command line usage.

## 2.3 Terminology

A **file** is an unit of digital data storage that can contain a variety of information, such as text, images or even programs. It is identified by an unique name and an extension, which indicates the file's format or type. Files are organized within a file system, allowing users to create, access, modify, and delete them.

A **folder** is a container used in file systems to store and organize files and other folders. Folders provide a hierarchical structure that allows users to group and manage files, making it easier to locate and navigate through data.

A **directory** is the more technical term, while **folder** is a user-friendly term for the same concept. In practice, the two terms are often used interchangeably, and many operating systems have adopted the term "folder".

The **"command-line interface (CLI)"** is a text-based way to interact with computer programs. Most people primarily use a Graphical User Interface (GUI) to work with their programs, but historically, text-based interfaces were more common. Often, the GUI can only access a limited set of features available through the CLI and some programs don't even have a GUI and can only be used through the command line. For many years, the command line was the primary way to interact with computers. Before GUIs like the ones we use today, early computer users communicated with their machines by typing text-based commands. This text-based interaction allowed them to perform various tasks, from running programs to managing files and directories.

**"Terminal"** is short for "terminal emulator", which is a program that mimics a physical device called a terminal. Terminals were used to connect to mainframe computers in the early days of computing. The terminal program creates the window with the dark background, light text, and cursor.

A **"shell"** is a program that provides the command line to your operating system. It allows you to perform various tasks that your operating system offers, such as managing files, running and stopping programs, and changing system settings, all through text commands. When you "open a terminal", the program initially communicating with the terminal is a shell. There are multiple shell options available for different operating systems, with the most popular one being the "Bourne Again Shell" or "Bash".

The terms "shell", "command line", "command-line interface (CLI)", and "terminal" are used interchangeably and generally refer to the same thing: a simple window with a dark background and light text, showing a prompt and a blinking cursor. In movies, when people hack into computers, they often type rapidly in this window.

## 2.4 Finding the command line

This section can you help you finding the command line on your system. Click on the tab for your operating system for detailed information.

### 2.4.1 MacOS

You can access your default shell through the Terminal program located within the Utilities folder: There are (at least) three different ways to get there:

- In the Finder, go to `Applications` > `Utilities` > `Terminal.app`
- In the Finder, select the `Go` menu > `Utilities` > `Terminal.app`
- Activate the Mac `Spotlight` search function, type `Terminal` in the search bar, and press the `Return` key.

You may want to keep Terminal in your Dock to find it more easily.

**Mac Terminal: Bash or Zsh?**

On a Mac computer running macOS Mojave or earlier versions, the default Unix Shell is Bash. Unix being a family of operating systems serving as the basis for Linux and macOS with the inclusion of a command line interface (or Unix Shell) as a key feature. For a Mac computer operating on macOS Catalina or later releases, the default Unix Shell is Zsh. Bash and Zsh are both programs, which are command-line interfaces for interacting with an operating system. Zsh is often considered better than Bash for its more advanced features and improved scripting capabilities. It also has extensive community-contributed plugins and themes. However, the change to Zsh was not a drastic shift for most users, since there a few practical differences.
To verify if your machine is configured to use a shell other than Bash, you can type `echo $SHELL` in your Terminal window. If the output is `/bin/bash` or a similar path, it means your default shell is set to Bash. This is the most common shell on Unix-based systems, including macOS. If the output is `/bin/zsh` or a similar path, it means your default shell is set to Zsh. If the output is a different path, it indicates that your shell is configured to use a different shell program. If your machine is configured to use a shell other than Bash, you can switch to Bash by opening a terminal and typing `bash`.

**Which macOS version do I have installed?**

1. In the upper-left corner of your screen, click on the  logo.
2. Select “About This Mac”: A dropdown menu will appear. Choose “About This Mac”.
3. View macOS Version: In the “About This Mac” window, you will see information about your Mac, including the macOS version. The version number will be listed under the “Overview” tab. For example, it might say “macOS Big Sur” or “macOS Monterey”, along with the version number (e.g., “Version 11.4”).

### 2.4.2 Windows

By default, Windows operating systems do not come with a pre-installed Unix Shell program. In this guide, we recommend using an emulator provided by Git for Windows, which grants you access to both Bash shell commands and Git.

For an installation guide, you can check out the chapter on installation and setup. Once you have installed Git for Windows, you can easily open a terminal by launching the program called Git Bash from the Windows start menu.

### 2.4.3 Linux

Open a fresh Terminal.

### 2.4.4 JupyterHub

If it is not possible for you to access a command line interface using your personal computer, you can use a remote solution with JupyterHub. JupyterHub is a web-based platform that allows multiple users to access an interactive computing environment on a shared server.

You can access a remote environment, using your Universität Hamburg login credentials here. Simply click on the link and log in.

Once you are logged in, you should be able to view the remote environment and create files using different programs like R, Python or Java.

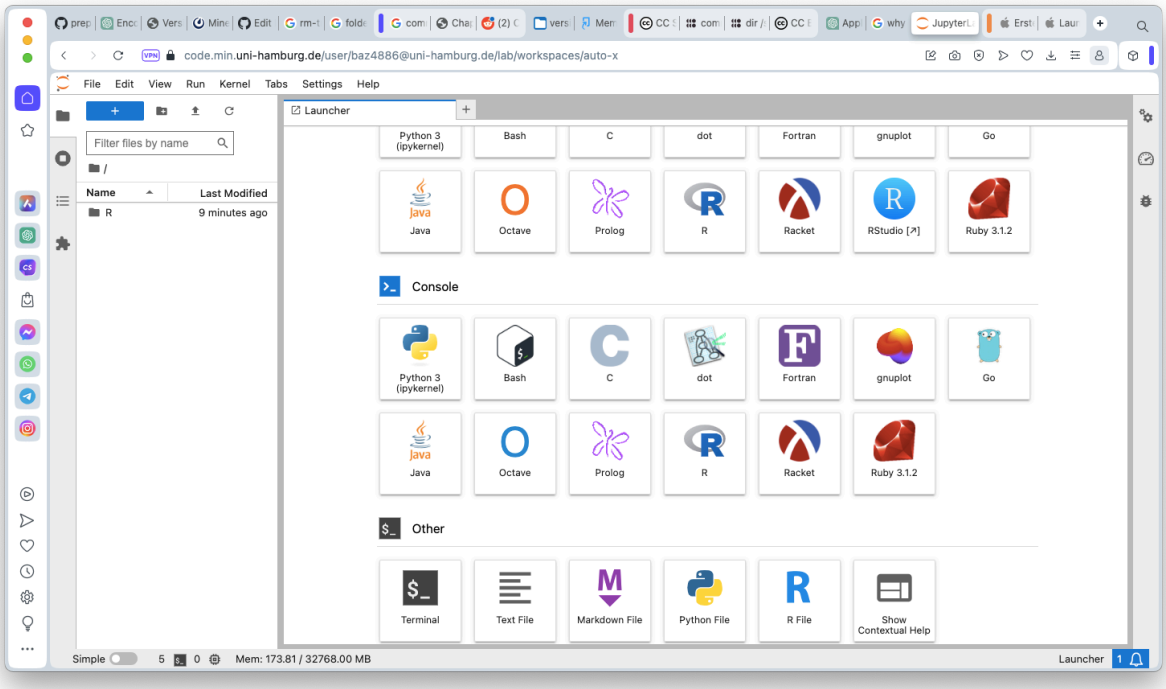



Figure 2.1: Screenshot of JupyterHub, showing the start screen.

If you scroll down in the “launcher” window, there should be the option to open a terminal window. Your remote environment will be mostly empty, but you can move around and manage files just like you could on your own computer.

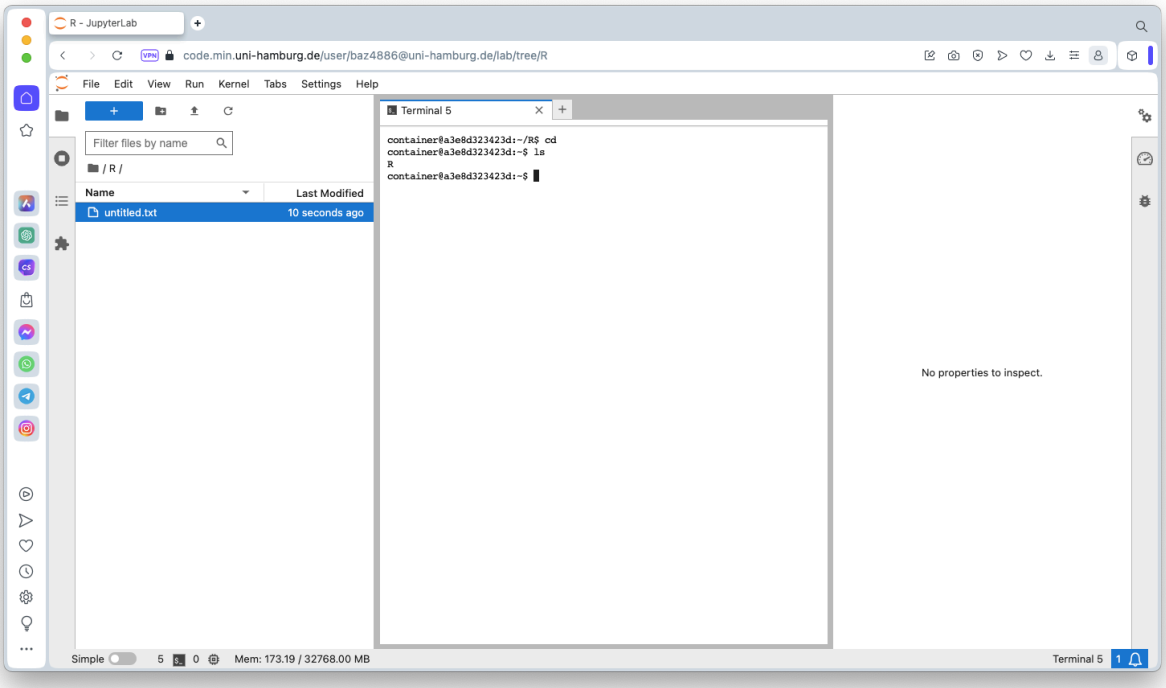



Figure 2.2: Screenshot of JupyterHub, showing the terminal.

## 2.5 Opening the command line

When you open the shell, you'll see a prompt that tells you the shell is ready for input. Your terminal interface will usually show a message similar to the following prompt:

```
$
```

In the shell, the prompt is usually represented by the symbol `$` but it can be different sometimes. In our examples, we'll use `$` as the prompt. The important thing to remember is **to not type the prompt, `$`, itself when entering commands**. Only type the command that comes after the `$` prompt. For this reason, we remove the `$` prompt from all code examples in this guide. Also, remember to press the `Enter` key after typing a command to execute it.

After the prompt, you'll see a text cursor that shows where you can type. The cursor can be a flashing or solid block, or it can be an underscore or a pipe symbol. You may have seen a similar cursor in a text editing program.

Keep in mind that your prompt may look slightly different. Usually, in popular shell environments, your username and the host name appear before the `$` symbol. For example, your prompt might look like this:

```
wittkuhn@lip-osx-005509:~$
```

## 2.6 Navigating the file system

### 2.6.1 Introduction

You might be familiar with using your mouse to click through folders in your file browser, or perhaps you use arrow keys and clicks in the Finder (on macOS) or File Explorer (on Windows) to manage your files.

The command line offers a different, one could say less intuitive, but very efficient and precise way to interact with your computer's file system. Instead of relying on a graphical user interface, you use text-based commands to perform tasks like exploring the contents of folders, creating and deleting files, and moving between folders.

### 2.6.2 Directory and file structure

In a file system, directories (or folders) and files are organized hierarchically to enable the storage and retrieval of data. Each level in the hierarchy is called a "level" or "depth". In the context of file systems, the hierarchy of directories is often described as a parent-child relationship and folders are therefore often referred to as "parent directories" and "child directories" (for details, see the box below).

The **root directory** is the highest level in the hierarchy. On Unix-based systems (including macOS), the root directory is denoted by `/`. On Windows, it is typically denoted by a drive letter followed by `\` (e.g., `C:\`). While the fundamental concepts of directories and files are consistent across operating systems, there are some differences in folder structures between macOS (Unix-based) and Windows:

What are "parent" and "child" directories?

In the context of file systems, a "parent directory" and a "child directory" describe the relationship between directories (or folders).

- **Parent Directory:** This is the directory that is "higher up" in the directory hierarchy and contains one or more "child directories."
- **Child Directory:** A child directory is located "inside" or "below" a parent directory. It is contained within the parent directory and is considered subordinate to it, much like a child in a family is part of the larger family unit.

```
parent $ tree
.
   child1
   child2

3 directories
```

#### 2.6.2.1 Windows

Each drive (e.g., `C:`, `D:`) has its own root directory, and it is denoted by the drive letter followed by `\` (e.g., `C:\`). Windows uses `\` as the directory separator. The home directory is often `C:\Users\username\`. An example path on Windows might be:

```
C:\Users\username\Documents\file.txt`
```

#### 2.6.2.2 macOS

On macOS the root directory is denoted by `/`, which is also used as the directory separator. The home directory for a user is typically `/Users/username/`. An example path on Mac OS could be:

```
/Users/username/Documents/file.txt
```

### 2.6.3 Current working directory

The current **working directory** is the location within your file system where a user or an application is presently performing file operations or running commands. The command `pwd` (short for "print working directory") will display the current directory that the shell is currently focused on in your file system.

```
pwd
```

👁 Click here to see the output of this command executed in the project directory of this guide.

```
/Users/wittkuhn/edu/version-control-book
```

### 2.6.4 Command not found

If the shell cannot find the program you typed as a command, it will show an error message. This can happen if you made a mistake when typing the command or if the program for that command is not installed on your system. For example, if you made a mistake when typing the `pwd` command, and accidentally entered `pws`, the following error message might appear:

```
pws: command not found
```

**Listing 2.1** `Absolute path`

```
wittkuhn@lip-osx-005509:~$ cd /Users/wittkuhn/Downloads
wittkuhn@lip-osx-005509:~/Downloads$ pwd
/Users/wittkuhn/Downloads
```

**Listing 2.2** `Relative path`

```
wittkuhn@lip-osx-005509:~$ cd Downloads
wittkuhn@lip-osx-005509:~/Downloads$ pwd
/Users/wittkuhn/Downloads
```

### 2.6.5 Moving between folders

The `cd` command is used to change the current working directory (or "move" to a different directory in your file system). To use it, type `cd` followed by an argument, which is the path of the folder you want to navigate to. The path can be either *absolute* or *relative*.

For example, if you are on macOS and your current working directory is your home directory (for example, `/Users/wittkuhn`), you can switch to the `/Downloads` folder by using either an absolute or relative path with the `cd` command:

You may notice that the prompt has been updated to show the new working directory. If a path starts with a forward slash (`/`), it is considered an *absolute* path. Paths that do not start with a forward slash are considered *relative* to the current working directory.

Note, that if a path contains a space, it needs to be enclosed in quotes to be recognized by the command line. Here is an example of a path with a space:

```
/Users/User/Downloads/Directory with Space/
```

If you attempt to use this path in a command without quotes, the command interpreter would understand it as three separate arguments since in command-line interfaces, spaces are typically used to separate different arguments or parameters. Navigating to this absolute path with spaces would look like this:

```
cd '/Users/User/Downloads/Directory with Space/'
```

While the command line can handle paths (or files) with spaces in this way, we **recommend to avoid spaces** in folder and file names altogether, especially if you use the command line frequently.

### 2.6.6 Taking shortcuts

#### 2.6.6.1 Tilde: ~

The tilde symbol (~) provides a convenient and abbreviated way to refer to the user's home directory, enabling us to easily construct paths relative to that directory. To quickly change the current working directory to your user's home directory anywhere on your file system, we can use `cd ~` which will transport us back to the user's home directory.

```
wittkuhn@lip-osx-005509:~/Downloads$ cd ~
wittkuhn@lip-osx-005509:~$
```

#### 2.6.6.2 Dot: .

Another symbol with a special meaning in paths, aside from ~, is the dot (`.`). It serves as a shorthand representation for the current working directory, and using it in a path means the reference remains within the same directory.

```
wittkuhn@lip-osx-005509:~$ cd .
wittkuhn@lip-osx-005509:~$
```

In turn, these two commands are equivalent:

```
wittkuhn@lip-osx-005509:~$ cd ./Downloads
```

```
wittkuhn@lip-osx-005509:~$ cd Downloads
```

#### 2.6.6.3 Dot Dot: ..

The `..` symbol is a shorthand representation for the parent directory of the current working directory. It can be used repeatedly, allowing `cd ..` to move up one folder and ‘`cd ../..` to move up two folders. This is useful for creating relative paths to other branches within a directory tree.

For instance, by executing the following sequence of commands, we will navigate horizontally (or sideways) to a folder located at the same hierarchical level as the current working directory:

```
wittkuhn@lip-osx-005509:~$ cd ~                                    ①
wittkuhn@lip-osx-005509:~$ cd Downloads                            ②
wittkuhn@lip-osx-005509:~/Downloads$ cd ../Documents               ③
wittkuhn@lip-osx-005509:~/Documents$ pwd                           ④
/Users/wittkuhn/Documents
```

① Move to the user's home directory using the ~ shorthand.
② Move to the `Downloads` folder.
③ Move sideways to the `Documents` folder.
④ Display the current working directory.

> **Using Command History Navigation**
>
> You can access your command history by using the up and down arrow keys on your keyboard to cycle through your last used commands. To execute a command, hit Enter, or you can edit it as needed before execution. This makes it easier to find and reuse specific commands. And is also the reason why the up arrow key is among the earliest that needs replacement on software developers' keyboards.

### 2.6.7 Autocompletion

While using the command line, instead of typing out the full command or path, you can press the ⇥ Tab key to automatically complete the rest. Begin typing a command or the initial letters of a file or directory, for example:

```
cd ~/Doc
```

Once you've typed a portion of the command or path, press the ⇥ Tab key. The command-line interface will attempt to autocomplete the command or path based on what matches your input. In this case for example:

```
cd ~/Documents/
```

If there are multiple possibilities, pressing ⇥ Tab again will cycle through the available options. Autocompletion also works for system commands. For example, typing `ls` and pressing ⇥ Tab might complete it to `ls -l` or `ls -a`. Autocompletion also works for the Git commands, that will be introduced in the following chapters of this book.

### 2.6.8 Command history

Using the `Up` and `Down` arrow keys on your keyboard allows you to quickly access and reuse commands without retyping them.

After executing one or multiple commands, pressing the `Up` arrow key retrieves the most recent command. If you've navigated back using the `Up` arrow, pressing the `Down` arrow key moves you forward through the command history. You can continue to press `Up` and `Down` to cycle through the executed commands.

The primary benefit of this is time efficiency. Instead of typing out the entire command, especially for lengthy or complex commands, you can retrieve and modify previous commands quickly. It also minimizes the chances of introducing typos or syntax errors when reusing commands.

### 2.6.9 Clearing the command line

The `clear` command is used to clear the content of your terminal, providing a clean and empty slate. It helps improve visibility by removing previous commands, outputs, and clutter from your terminal window.

To clear the content of the terminal screen, simply type and execute the `clear` command:

```
clear
```

Press `Enter` and the terminal screen will be cleared, leaving only a new prompt at the top of the window. After using `clear`, you can continue working in a fresh and uncluttered terminal environment.

## 2.7 List files and folders

The `ls` command is used to list the files and folders in a specified directory or the current working directory. If you run this command, you will see the contents of your current working directory.

```
ls
```

👁 Click here to see the output of this command executed in the project directory of this guide.

```
CITATION.cff
LICENSE
Makefile
README.md
_affiliations.yml
_authors.yml
_book
_extensions
_metadata.yml
_quarto.yml
_variables.yml
chapters
cheatsheet.json
chicago-author-date-note.csl
contents.qmd
exercises
functions.R
index.qmd
misc
objectives
plausible.html
references.bib
renv
renv.lock
```

```
static
version-control-book.Rproj
```

However, the `ls` command provides various options, such as flags and arguments, to enhance its functionality beyond simple file listing. To provide additional instructions to a command, we can pass flags and arguments by typing them after the command name.

### 2.7.1 List files in a specific directory

For example, we can list the contents of the `/chapters` folder in the project directory of this guide:

```
ls chapters
```

👁 Click here to see the output of this command executed in the project directory of this guide.

```
branches.qmd
command-line.qmd
command-line.rmarkdown
first-steps-git.qmd
github-intro.qmd
gui.qmd
installation.qmd
intermediate-commands.qmd
intro-version-control.qmd
issues.qmd
project-management.qmd
rewriting-history.qmd
setup.qmd
tags-and-releases.qmd
```

In this case, `chapters` is an argument to the `ls` command.

### 2.7.2 Listing hidden files

Hidden files, often starting with a dot in their names, are files that remain mostly concealed from the regular view in GUIs. They can serve different purposes, such as storing configuration settings, sensitive data, or temporary files. Hidden files can be accessed through the command line or by changing settings in the file manager to make them visible. It's important to be cautious when modifying or deleting hidden files, as they can impact the proper functioning of the operating system and applications. This is also typically the reason for these files being hidden. To view hidden files, we can use the `-a` (for "all") flag in the `ls` command. For example, we can list hidden files in the project directory:

```
ls -a
```

👁 Click here to see the output of this command executed in the `/chapters` directory of this guide.

```
.
..
.DS_Store
.Renviron
.Rhistory
.Rprofile
.Rproj.user
.all-contributorsrc
.codespellrc
.git
.github
.gitignore
.luarc.json
.quarto
.zenodo.json
CITATION.cff
LICENSE
Makefile
README.md
_affiliations.yml
_authors.yml
_book
_extensions
_metadata.yml
_quarto.yml
_variables.yml
chapters
cheatsheet.json
chicago-author-date-note.csl
contents.qmd
exercises
functions.R
index.qmd
misc
objectives
plausible.html
references.bib
renv
renv.lock
static
version-control-book.Rproj
```

### 2.7.3 Interlude: What is the difference between a flag and an argument?

#### 2.7.3.1 Overview

The difference between a flag and an argument is that a **flag is used to modify the behavior** of a command, while an **argument provides additional input** or information to the command.

#### 2.7.3.2 What is a flag?

A flag is typically a single character or a combination of characters preceded by a hyphen (-) or double hyphen (--). It is used to enable or disable specific options or features of a command. Flags are optional and are used to customize the behavior of the command.

#### 2.7.3.3 What is an argument?

An argument is the actual input or information that is passed to a command to perform a specific action. It can be a value, a file name, a directory path, or any other data required by the command to complete its task. Arguments are often positioned after the command and any flags, and they can be mandatory or optional depending on the command's requirements.

#### 2.7.3.4 Multiple single-letter flags

You can combine multiple single-letter flags by using a single dash. For example, to instruct the `ls` command to list *all* files (including hidden files) in a *long* format that is easy to read for *humans*, you can use the following combination of flags:

```
ls -alh
```

👁 Click here to see the output of this command executed in the project directory of this guide.

```
total 408
drwxr-xr-x  41 wittkuhn  staff   1.3K Feb 28 20:39 .
drwxr-xr-x  28 wittkuhn  staff   896B Feb 16 13:52 ..
-rw-r--r--@  1 wittkuhn  staff   6.0K Feb 27 20:48 .DS_Store
-rw-r--r--   1 wittkuhn  staff    36B Aug 10  2023 .Renviron
-rw-r--r--   1 wittkuhn  staff    22K Feb 28 14:05 .Rhistory
-rw-r--r--   1 wittkuhn  staff    48B Aug 10  2023 .Rprofile
drwxr-xr-x   4 wittkuhn  staff   128B Jun 22  2023 .Rproj.user
-rw-r--r--   1 wittkuhn  staff   1.5K Jan 12 09:44 .all-contributorsrc
-rw-r--r--   1 wittkuhn  staff    76B Jan 12 09:44 .codespellrc
drwxr-xr-x  15 wittkuhn  staff   480B Feb 28 20:39 .git
drwxr-xr-x   4 wittkuhn  staff   128B Jun 29  2023 .github
-rw-r--r--   1 wittkuhn  staff   143B Feb 28 20:36 .gitignore
-rw-r--r--   1 wittkuhn  staff   821B Jul 13  2023 .luarc.json
drwxr-xr-x   8 wittkuhn  staff   256B Feb 28 20:39 .quarto
```

```
-rw-r--r--   1 wittkuhn  staff   925B Feb 28 20:36 .zenodo.json
-rw-r--r--   1 wittkuhn  staff   992B Feb 28 20:36 CITATION.cff
-rw-r--r--   1 wittkuhn  staff    20K Aug 11  2023 LICENSE
-rw-r--r--   1 wittkuhn  staff   728B Feb 28 20:36 Makefile
-rw-r--r--   1 wittkuhn  staff   6.6K Feb 28 20:36 README.md
-rw-r--r--   1 wittkuhn  staff   3.0K Jun 29  2023 _affiliations.yml
-rw-r--r--   1 wittkuhn  staff   362B Feb 28 20:36 _authors.yml
drwxr-xr-x   2 wittkuhn  staff    64B Feb 28 20:39 _book
drwxr-xr-x   3 wittkuhn  staff    96B Jun 29  2023 _extensions
-rw-r--r--   1 wittkuhn  staff    86B Feb 28 20:36 _metadata.yml
-rw-r--r--   1 wittkuhn  staff   4.4K Feb 28 20:36 _quarto.yml
-rw-r--r--   1 wittkuhn  staff   1.7K Feb 28 20:18 _variables.yml
drwxr-xr-x  17 wittkuhn  staff   544B Feb 28 20:39 chapters
-rw-r--r--   1 wittkuhn  staff   5.3K Feb 28 20:36 cheatsheet.json
-rw-r--r--   1 wittkuhn  staff    22K Aug 10  2023 chicago-author-date-note.csl
-rw-r--r--   1 wittkuhn  staff   671B Feb 28 20:36 contents.qmd
drwxr-xr-x  14 wittkuhn  staff   448B Feb 28 20:18 exercises
-rw-r--r--   1 wittkuhn  staff   1.7K Feb 28 20:36 functions.R
-rw-r--r--   1 wittkuhn  staff   5.0K Feb 28 20:36 index.qmd
drwxr-xr-x   8 wittkuhn  staff   256B Feb 28 20:36 misc
drwxr-xr-x  14 wittkuhn  staff   448B Feb 28 20:36 objectives
-rw-r--r--   1 wittkuhn  staff   118B Jun 29  2023 plausible.html
-rw-r--r--   1 wittkuhn  staff    11K Feb 27 20:49 references.bib
drwxr-xr-x   7 wittkuhn  staff   224B Aug 10  2023 renv
-rw-r--r--   1 wittkuhn  staff    15K Aug 10  2023 renv.lock
drwxr-xr-x  78 wittkuhn  staff   2.4K Feb 28 19:28 static
-rw-r--r--@  1 wittkuhn  staff   211B Feb 28 14:05 version-control-book.Rproj
```

#### 2.7.3.5 Getting help

#### 2.7.3.6 Windows (GitBash)

If you want to know what flags and arguments a command supports, most commands provide usage information when you use the flag `--help`. For example, you can use the `ls` command together with the `--help` flag:

```
ls --help
```

This will list all possible flags you can use together with the `ls` command.

#### 2.7.3.7 macOS

On Mac OS you can not use the `--help` flag. Instead you can use the `man` command, in combination with your desired command. This will display the manual page for a specified command. You can navigate through the manual pages using the arrow keys, and you can exit by pressing `q`. For example:

```
man ls
```

## 2.8 Manipulating the file system

The command line offers commands for creating, moving, and deleting folders and files.

### 2.8.1 Creating a folder

Let's use the command line to create a new project folder as an example: To make a directory, we'll use the `mkdir` command.

**Note:** When creating new directories, consider their location carefully. Consider deleting the newly created directories afterwards to avoid clutter.

The path of the directory we want to create is given as an argument to the command. If we want to create a directory in the current folder, we just need to use its name, as the path is assumed to be relative.

```
mkdir my-project
```

Make sure the project folder is present by using the `ls` command, and then move into it by using the `cd` command.

```
ls
cd my-project
```

### 2.8.2 Creating multiple folders

Next, we will proceed to create multiple additional folders within the `my-project` project folder. The `mkdir` command allows us to create multiple directories simultaneously by providing multiple arguments, each representing a separate folder. For example, for a data analysis project we might create the following three folders:

```
mkdir data results scripts
```

### 2.8.3 Creating files

Let's create some files in the `my-project` project folder. To do this, you can use your favorite text editor or the command line. In the command line, you can use the `touch` command to create a new empty file. The file doesn't need to contain anything. For example, to create an empty file called `file.txt`, we can use the following command:

```
touch file.txt
```

### 2.8.4 Opening files

To open files, you can use your preferred text editor but also the command line. On the command line, you can use the `open` command on macOS or the `start` command on Windows using Gitbash.

#### 2.8.4.1 Windows (GitBash)

```
start file.txt
```

#### 2.8.4.2 macOS

```
open file.txt
```

These commands will open the specified file in the default application associated with the file type.

### 2.8.5 Writing short text into files

The `echo` command can be used to write text to a file, directly from the command line. Using `echo` in combination with >> adds content to an existing file. For example, to add text to the `file.txt` file:

```
echo "This is a project for demonstration purposes." >> file.txt
```

This adds the provided text to the existing content of `file.txt`.

### 2.8.6 Viewing file content

If you want to quickly view the content of a file without opening it, you can do so in the command line using the `cat` command. To display the content of a file, simply use `cat` followed by the file name. For example:

```
cat file.txt
```

This command will output the entire content of the `file.txt` file to the terminal.

👁 Click here to see the output of the `cat` command applied to the `.gitignore` file in the project directory of this guide.

```
/.quarto/
/_book/
.Rproj.user
/env/
.Rhistory
static
/.luarc.json
.DS_Store
*.aux
*.log
*.pdf
*.tex
*.toc
*.fdb_latexmk
*.xml
*.bcf
*.blg
*.bbl
```

### 2.8.7 Moving folders and files

Let's say we want to move the `file.txt` file to the `/scripts` folder. We can use the `mv` command to move files and folders. The `mv` command requires two arguments: the first is the files or folders to be moved, and the second is the path where they should be moved. To move the `file.txt` file, the command would look like this:

```
mv file.txt scripts
```

### 2.8.8 Renaming files and folders

Suppose we don't like the folder name `scripts` because `file.txt` is actually a documentation file, not a script. We can use the `mv` command to rename files and folders by moving them to a different location in the file system, for example `docs`, like this:

```
mv scripts docs
```

### 2.8.9 Wildcards

Now, let's learn how to use wildcards to perform actions on multiple files or folders at the same time. A wildcard is denoted by the symbol `*`, and it helps us select file system arguments that match certain patterns. For example, suppose we wish to display a list of all files of a certain file type (here, all files with the `.yml` file extension) in the current directory:

```
ls -alh *.yml
```

👁 Click here to see the output of this command executed in the project directory of this guide.

```
-rw-r--r--  1 wittkuhn  staff   3.0K Jun 29  2023 _affiliations.yml
-rw-r--r--  1 wittkuhn  staff   362B Feb 28 20:36 _authors.yml
-rw-r--r--  1 wittkuhn  staff    86B Feb 28 20:36 _metadata.yml
-rw-r--r--  1 wittkuhn  staff   4.4K Feb 28 20:36 _quarto.yml
-rw-r--r--  1 wittkuhn  staff   1.7K Feb 28 20:18 _variables.yml
```

💡 What does this output mean?

The first column (`-rw-r--r--`) represents the file permissions. In Unix-like systems, it shows who can read, write, or execute the file. In this case, the files are readable and writable by the owner (`rw-`), readable by the group (`r--`), and readable by others (`r--`).
The second column (`1`) indicates the number of hard links to the file.
The third column (`runner`) is the username of the file's owner. Here, the username is `runner` because the this online guide is build automatically on GitHub using GitHub Actions and `runner` refers to the computing environment where the build process is executed. If you would build the book on your computer, it would show your username.
The fourth column (`docker`) is the group associated with the file. Here, the group name is `docker` which again refers to the computing environment where this online guide is built.
The fifth column shows the size of the file. In this case there are just small textfiles included with the sizes ranging from `97B` to `3.0K`.
The next column (for example, `Jan 18 20:39`) display the date and time when the file was last modified.
The last column (for example, `_affiliations.yml` etc.) shows the name of each file.

By using the wildcard `*`, we can generate a list of all files ending with `.yml`. Similarly, we can apply the wildcard at the end of a path to match items starting with a specific letter, such as folders beginning with `i`:

```
ls -alh i*
```

👁 Click here to see the output of this command executed in the project directory of this guide.

```
-rw-r--r--  1 wittkuhn  staff   5.0K Feb 28 20:36 index.qmd
```

### 2.8.10 Removing files and folders

To delete files and folders, you can use the `rm` command. Provide the path of the files or folders that you want to delete as the argument. However, if you want to remove a folder, you need to include the `-r` (or `--recursive`) flag.

For example, to remove the `docs` folder, you would use the following command:

```
rm -r docs
```

> ⚠ **Warning: Be careful when using the `rm` command!**
>
> When you use the `rm` command to delete files, they are **removed entirely (!!!)** from your system and cannot be retrieved from a "Recycle Bin" or "Trash" as when you delete files using the file browser!

### 2.8.11 Combining commands

Combining commands in the command-line using semicolons `;` is a way to execute multiple commands sequentially on a single line. Each command is separated by a semicolon, and they are executed one after the other, regardless of the success or failure of the previous command.

For example, let's say you want to (1) create a new directory, (2) move into that directory, and then (3) list the files in that directory, all in one go. You can use semicolons to combine these commands like this:

```
mkdir new_directory ; cd new_directory ; ls
```

This will (1) create a new directory called `new_directory`, (2) change the current working directory to `new_directory` and (3) list the files in this new directory.

## 2.9 Summary

In this lesson, we have achieved the following:

1. Clarified some of the technical terms associated with the command line.
2. Explored the components of command-line commands: paths, arguments, and flags.
3. Gained knowledge about relative and absolute paths, including convenient shortcuts like `~`, `.` and `..`.
4. Witnessed the functionality of wildcards.
5. Acquired familiarity with several essential shell commands.

It's important to note that what we covered here only scratches the surface of what can be accomplished using the shell. The primary aim was to provide useful concepts that help when learning Git. If you're interested in delving deeper, you may find a more comprehensive introduction at one of the following resource.


Figure 2.3: “Hackerman” meme found on Reddit.com

## 2.10 Acknowledgements & further reading

We would like to express our gratitude to the following resources, which have been essential in shaping this chapter. We recommend these references for further reading:

| Authors | Title | Website | License | Source |
|---|---|---|---|---|
| Millman et al. [14] | Teaching Computational Reproducibility for Neuroimaging | | CC BY 4.0. Website: http://dx.doi.org/10.3389/fnins.2018.00727 | |
| Milligan and Baker [13] | Introduction to the Bash Command Line | NA | NA | NA |
| McBain [12] | Git for Scientists | | CC BY-SA 4.0 | |
| Capes et al. [6] | swcarpentry/shell-novice: Software Carpentry: the UNIX shell | | CC BY 4.0 | |

## 2.11 Cheatsheet

| Command | Description |
|---|---|
| `pwd` | Display the current directory path |
| `cd <PATH>` | Change the current working directory to `<PATH>` |
| `cd ~` | Change the current working directory to the user's home directory |
| `cd ..` | Move up one folder |
| `cd ../..` | Move up two folders |
| `clear` | Clears content of of your terminal window |
| `ls` | List files and folders in the current working directory |
| `ls <PATH>` | List files and folders in `<PATH>` |
| `ls -a` | List all files (including hidden files) in the current working directory |
| `ls -alh` | List all files in a long format that is easy to read for humans |
| `[Command] --help` | Provides all possible flags for a specific command (on Windows) |
| `man [Command]` | Provides all possible flags for a specific command (on Mac OS) |
| `mkdir <FOLDER>` | Create a new folder, called `<FOLDER>` |
| `mkdir <FOLDER1> <FOLDER2>` | Create two separate folders, called and |
| `touch <FILE>` | Create a new empty file, called `<FILE>` |
| `open <FILE>` | Opens the file called (On macOS) |
| `echo 'example text' >> file.txt` | Writes 'example text' into `file.txt` |
| `cat <FILE>` | Displays the content of |
| `start <FILE>` | Opens the file called (On Windows) |

| Command | Description |
|---|---|
| `mv FILE.txt <FOLDER>` | Move `FILE.txt` into `<FOLDER>` |
| `mv <FOLDER_OLD> <FOLDER_NEW>` | Rename `<FOLDER_OLD>` to `<FOLDER_NEW>` |
| `ls -alh *.csv` | Use a wildcard to list all `.csv` files |
| `rm -r <FOLDER>` | Remove the folder `<FOLDER>` |

# 3 Installation

All beginnings are difficult: This chapter is about downloading and installing Git.

Take the quiz!

Learning Objectives
- ☐ Downloading and installing Git

## 3.1 Downloading Git

### 3.1.1 macOS

#### 3.1.1.1 Option 1: Terminal

Where is the Terminal?

The Terminal can be found at `/Applications/Utilities/Terminal.app` or by searching the Spotlight for "Terminal".
If you are unsure where to find the terminal, we recommend to read the "Command Line" chapter beforehand.

MacOS usually comes with a pre-installed version of Git that is accessible through the Xcode Command Line Tools. Open the Terminal and type:

```
git --version
```

If Git is installed, you should receive an output similar to:

```
git version 2.39.2 (Apple Git-143)
```

If Git is not installed you will get a prompt asking you if you wish to install it along with Xcode command line tools. You also may be asked for your administrator password.

Error message: `xcrun: error`

**Problem**
Did you receive the following error message when entering a `git` command into the Terminal?

```
xcrun: error: invalid active developer path (/Library/Developer/CommandLineTools), missing
```

**Solution**

```
xcode-select --install
```

This will open a dialogue box. Select “Install”, and it will download and install the Command Line Tools package and fix the problem.

#### 3.1.1.2 Option 2: Binary installer

Download and run the installer from: https://git-scm.com/download/mac. Hit `Continue` through the prompts without modifying the installation destination or other settings. You may be asked for your administrator password.

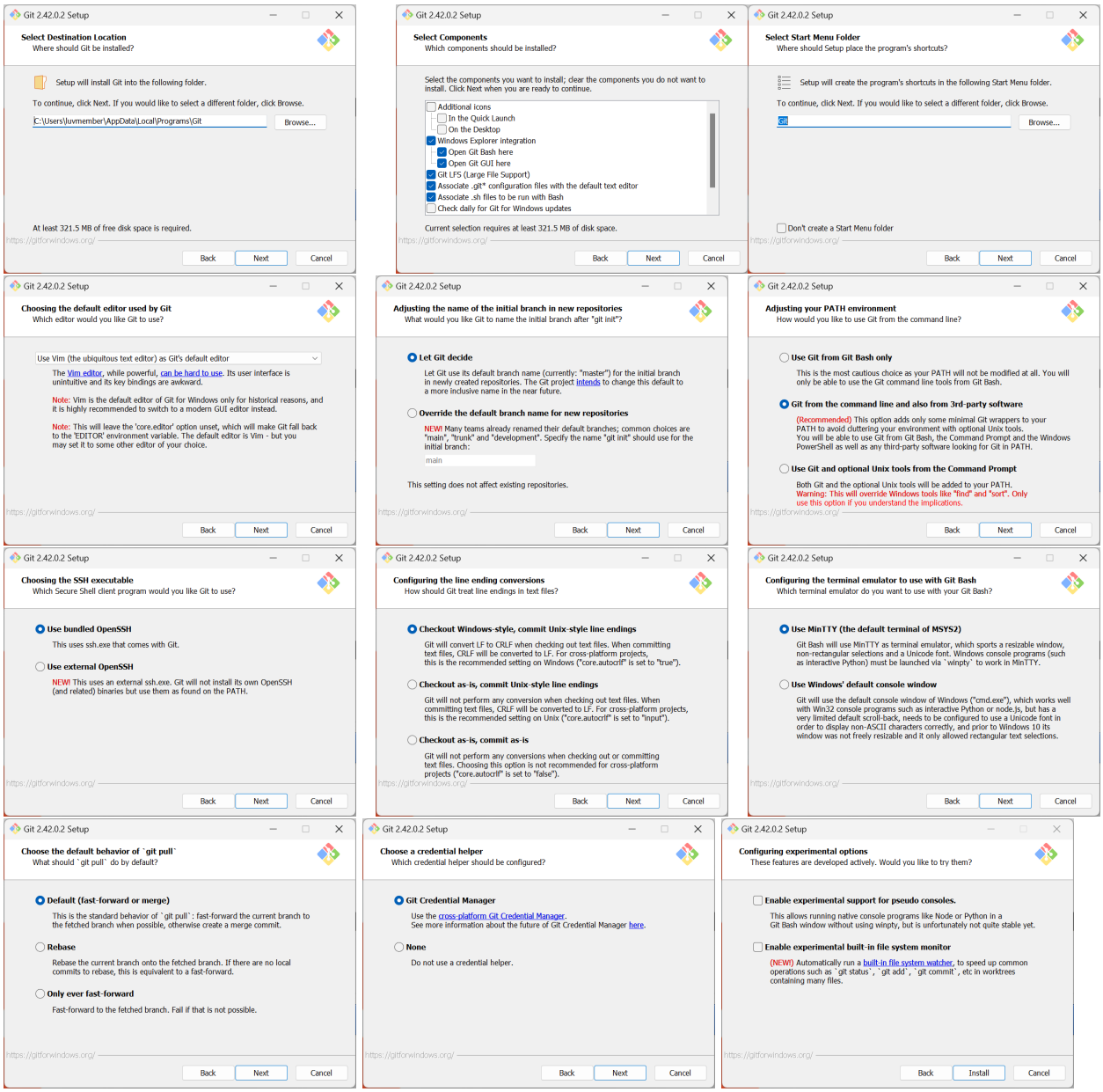
### 3.1.2 Windows

To use Git on a Windows computer, we recommend the installation of “Git for Windows”, which includes both the Bash shell and Git. You can download this program at gitforwindows.org.


Figure 3.1: Screenshot of gitforwindows.org

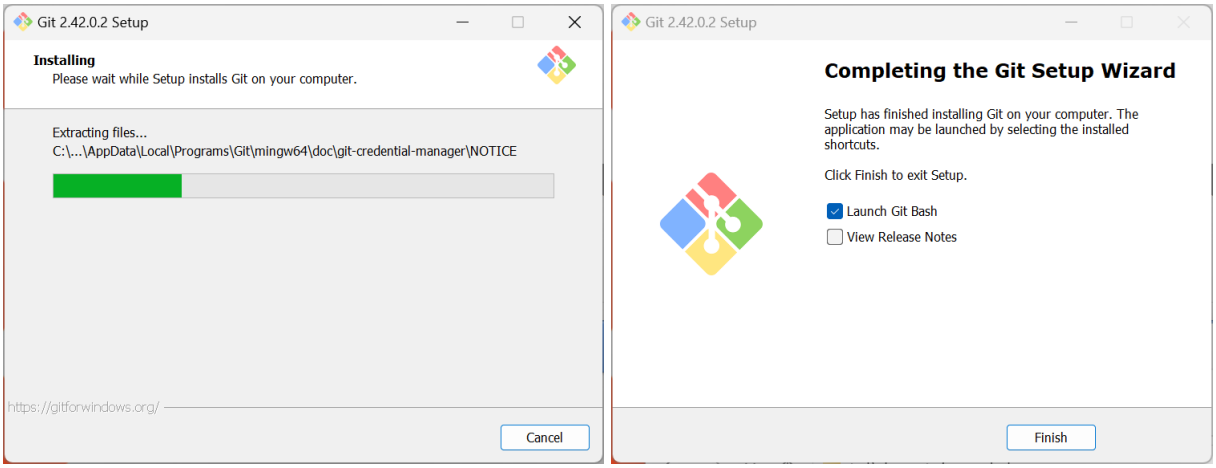
On the homepage, you should see a prominent download link.

Click on it to start downloading the installer. Once the installer is downloaded, double-click it to run the installation. During the installation process, you'll be presented with various options and settings. We recommend to leave all settings to their standard configuration, unless you have a specific need to e.g. change the installation destination.




Once you have finished selecting your preference, click the install button to begin the installation process.

The installer will copy the necessary files and set up Git and Git Bash on your system. Once the installation is complete, you'll see a "Finish" button and the option to directly launch a Bash window.




Now a Bash window opens up and you should be able to use the command line and Git!

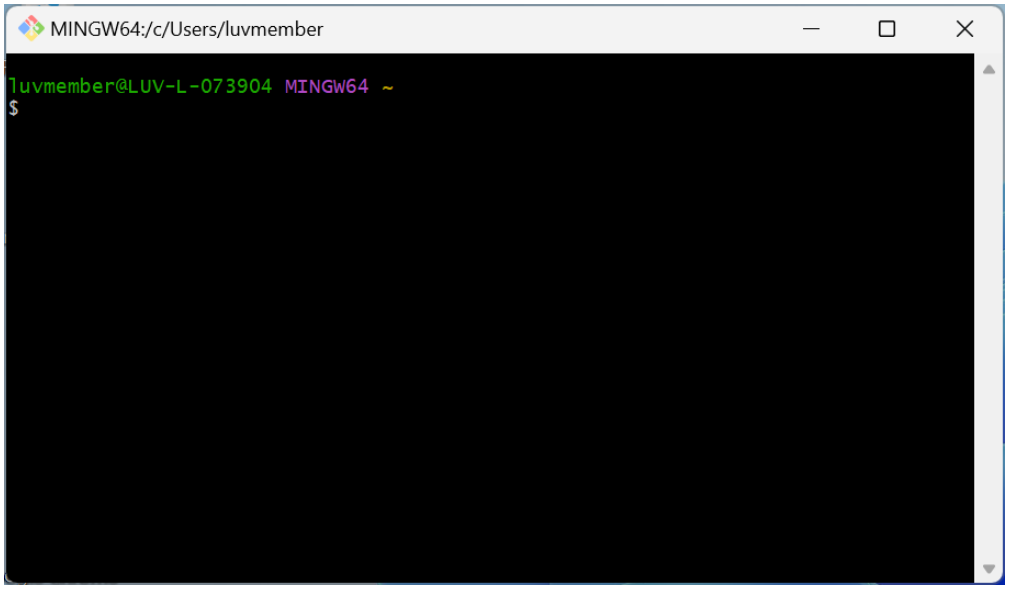



Figure 3.2: Screenshot of “Git Bash”

### 3.1.3 Linux

Check if you have git installed by opening your terminal and running:

```
git --version
```

If git is not installed, install it using your package manager, e.g., by running the following command in the terminal:

```
sudo apt-get install git
```

## 3.2 Acknowledgements and further reading

We would like to express our gratitude to the following resources, which have been essential in shaping this chapter. We recommend these references for further reading:

| Authors | Title | Website | License | Source |
|---|---|---|---|---|
| Chacon and Straub [7] | Pro Git | | CC BY-NC | |
| Capes et al. [6] | swcarpentry/shell-novice: Software Carpentry: the UNIX shell | | CC BY 4.0 | |
| Koziar et al. [10] | swcarpentry/git-novice: Software Carpentry: Version Control with Git 2023-05 | | CC BY 4.0 | |
| Bryan [5] | Happy Git and GitHub for the useR | | CC BY-NC 4.0 | |

# 4 Setup

In this chapter, we will configure Git!

Learning Objectives
- ☐ You know how to set up Git for the first time
- ☐ You have set up Git on your computer
- ☐ You understand the different Git configuration levels
- ☐ You know how to configure your username and email address in Git
- ☐ You have set up your preferred text editor when working with Git
- ☐ You can escape the command-line text editor Vim

## 4.1 Configuring Git

Once you installed Git, it is time to set it up. We will use the the command line to set up and configure Git. The `git config` command is used to get and set configuration. It allows you to customize Git according to your preferences and requirements. To look at all the different configuration commands you can use

```
git config
```

This chapter will focus on the most important and / or necessary configurations.

What are the different configuration levels?

Git configuration values can be stored at **three different configuration levels**:
**System-level configuration**: This level applies configuration values to every user on the system and all their repositories. To read or write configuration values at this level, you need administrative or superuser privileges.
**User-level configuration**: This level is specific to an individual user. Configuration values set at this level will be applied to all repositories owned by the user.
**Repository-level configuration**: This level is specific to a particular Git repository. The configuration values set at this level override values set at the system-level and user-level configurations for that specific repository. By default, Git reads from and writes to this file when you use the `git config` command without any additional options.

### 4.1.1 Identifying yourself

Before you start using Git, it's important to set your username and email address. This information is crucial because Git uses it to identify the author of each change you track with Git. By associating your changes with your username and email, it allows others (including future you) to see who made the changes which facilitates transparency and collaboration within a project. You will be able connect Git to remote repositories like GitHub, using this email address. So it makes sense to set your email address to the one you use (or will use) to create a GitHub account.

Setting your username and email address is a one-time setup process, but you can always update them later, if needed. To perform this configuration, you can use the command line:

```
git config --global user.name "Jane Doe"
git config --global user.email jane@example.com
```

Make sure to replace `"Jane Doe"` with your own username and `jane@example.com` with your own email address.

> FAQ: Which name should I use for Git?
>
> It can be your **GitHub username**, your **real name**, or something informative about you. This name will be attached to the changes you track with Git, so **choose something that makes sense** to others and yourself in the future.

### 4.1.2 Configure the default name of the initial branch

Git uses a concept called "branches" to enable working on different versions of the same project. (More on this will be explained later in the book.)

On most installations of Git, by default, the name "master" for the initial branch is used when creating a new repository. However, there has been a growing movement to transition to using "main" as the default branch name instead. One of the main motivations behind this change is to promote inclusivity and remove offensive terminology from the default Git workflow (you can read more about the reasons here or here). To configure the default branch name to "main" in Git, you can use the following command

```
git config --global init.defaultBranch main
```

## 4.2 Check your settings

You can verify if your configurations were set correctly using the following command:

```
git config --list
```

This should create a long output list, in which you should find your set configurations, for example:

```
user.name=Jane Doe
user.email=jane@example.com
init.defaultbranch=main
```

## 4.3 Calling for Help

The `--help` option in Git is a command that provides you with access to the built-in Git documentation and help resources. When you run a Git command followed by `--help`, Git displays information about that command, including its usage, available options, and a brief description of its functionality.

For example, the following command will open up a browser in the command line containing information about the `git config` command.

```
git config --help
```

To quit the browser use `q`.

While the command provides extensive documentation and information about Git commands, it is arguably not the most beginner-friendly resource.

## 4.4 Text editor

Sometimes, Git might open up a text editor inside the command line, and it can be annoying if you end up in an editor you don't like or don't know how to use. The standard text editor on macOS and when using "Git for Windows" is called Vim, which is widely known for being difficult to use. You can find more information about the challenges of using Vim (and some funny memes) here. To check out which text editor Git is using, you can use:

```
git config --get core.editor
```

If you have not changed your text editor yet, the output should be empty. This means that Git is using the command line´s default editor, which should be **Vim** if you are using a Mac or Git for Windows. To make sure, you can use:

```
git var GIT_EDITOR
```

If the output is: `vim` or `vi`, your systems default editor is indeed Vim/Vi. (Vim is an improved version of Vi, but there are virtually no differences between them for our usecase.)

In this situation, you can either pick up on the basics of Vim (see below) or simply change the text editor. Also note that you will typically not have to use the text editor in the command line a lot.

### 4.4.1 Vim Basics

To start Vim, open the command line and type `vim` followed by the filename of the file that you want to edit (for example, `vim file.txt`). In Vim, you have two main modes: "Insert" mode for typing and "Normal" mode for navigation and command execution. Inside Vimn to switch from "Normal" mode to "Insert" mode, press `i` for inserting text. To save your changes and exit Vim, press `Esc` to enter "Normal" mode, then type `:w` to save, and `:q` to quit. Vim's strength lies in its powerful commands for text manipulation, such as copying, pasting, and searching, which are executed in "Normal" mode.

For a more detailed and interactive tutorial, you can check out openvim.com.

### 4.4.2 Changing your text editor

If you want to change the standard text editor, to one you are more comfortable with, there a lot of alternatives like Nano, Visual Studio Code or Notepad++. If you have decided to use a different text editor you can use the following command (replace `editor-name` with the name of your preferred text editor):

```
git config --global core.editor editor-name
```

From now on, when Git requires you to interact with a text editor, it will use the one you have set. If you ever want to change your Git editor in the future, you can repeat the above steps and specify the new editor command or path.

## 4.5 Unsetting configuration

If you want to undo a configuration setting, you can use the `--unset` in the `git config` command. For example, to unset the global configuration of your preferred text editor (`core.editor`) you can use the following command:

```
git config --global --unset core.editor
```

## 4.6 Acknowledgements and further reading

We would like to express our gratitude to the following resources, which have been essential in shaping this chapter. We recommend these references for further reading:

| Authors | Title | Website | License | Source |
|---|---|---|---|---|
| Chacon and Straub [7] | Pro Git |  | CC BY-NC |  |

| Authors | Title | Website | License | Source |
|---|---|---|---|---|
| Capes et al. [6] | swcarpentry/shell-novice: Software Carpentry: the UNIX shell | | CC BY 4.0 | |
| Koziar et al. [10] | swcarpentry/git-novice: Software Carpentry: Version Control with Git 2023-05 | | CC BY 4.0 | |
| Bryan [5] | Happy Git and GitHub for the useR | | CC BY-NC 4.0 | |

## 4.7 Cheatsheet

| Command | Description |
|---|---|
| `git config` | Overview of config commands |
| `git config --global user.name` | Set Username |
| `git config --global user.email` | Set Email |
| `git config --global core.editor` | Set Editor |
| `git config --global init.defaultBranch main` | Set default branch name |
| `git config --list` | View set configurations |

# 5 First steps with Git

Exploring the most important basic commands you will need.

Take the quiz! UHH WS 23/24

Learning Objectives
- ☐ Initializing a Git repository
- ☐ Staging and committing changes
- ☐ Exploring the commit history
- ☐ Comparing versions
- ☐ Creating a `.gitignore` file

Exercises
To learn Git effectively, it makes sense to practice version control by implementing it on your own small project. For the purpose of this guide, we will start off with a small project that only involves plain-text files. You don't need to know any programming language like R or Python. So while reading this chapter and using the commands along, **your task** is to:
☐ Create a folder called `recipes` and initialize it as a Git repository. ☐ Create a plain text file called `recipes.txt` in this folder. ☐ **Add a recipe** in this file.
This can be the recipe for your favorite dish or an unusual AI-generated recipe. ☐ Stage and commit your changes to the file
Please **keep this project directory**! This guide will continue to use your recipe project as an example in following chapters.
There are several terms in these instructions that might be unfamiliar to you, for example "repository", "stage" or "commit". Don't worry, you will learn about what these terms mean in this chapter. Let's *git* started!

## 5.1 Creating a Git repository

First, we will create a so-called "repository" (or "repo", for short). A repository is a regular folder on your computer that is tracked by Git. The full history of commits of a project and information about who changed what, when is also stored by Git in this folder.

To create your first Git repository, you need to initialize a folder as a Git repository.

First, create an empty folder, for example using the command line:

```
mkdir foldername
```

Alternatively, you can create a folder using the graphical user interface of your file browser.

Next, navigate, using the Terminal, to the folder which you want to initialize as a Git repository. You can use the `cd` command to change directories. Once you are in the desired folder, run the `git init` command to initialize the folder as a Git repository.

```
git init
```

You should see an output similar to:

```
Initialized empty Git repository in /foldername/.git/
```

Congratulations on successfully initializing your first Git repository! 

Git is now able to track your file(s) and the changes you make to them in this folder. You only need to use `git init` once per folder.

You can also skip creating a folder manually and only use the `git init` command:

```
git init /Users/yourusername/Desktop/foldername
```

Git will create an already initialized folder at the path and with the name you use. Check your current working directory to be sure that you create the Git repository in the desired location. You can use both absolute and relative paths as arguments to the `git init` command to specify the location of your Git repository.

Git can now track every file you move, create or change in this folder. You can use this folder, like any other normal folder on you computer. The only difference is the tiny but powerful folder called `.git` within it, which stores the full history of files and relevant metadata of your project, thereby enabling the tracking of your project progress.

> **`.git` folder?**
>
> If you want to verify that Git is tracking your folder, you can look for the `.git` folder. To do this navigate to the correct folder and use `ls -a` to receive a list of hidden files. If the folder has been initialized as a Git repository, you should see a folder called `.git`. You probably should not manually modify this folder, since doing so can corrupt your repository.

> **Don't mess with the `.git` folder!**
>
> The `.git` folder in a Git repository contains all the essential information and configuration for the repository to function properly. Tampering with this folder can have serious consequences, as it can lead to data loss, corruption, or the inability to use the repository effectively. It's essential to only modify Git repositories through Git commands and established workflows to ensure the integrity and reliability of your project. In short: Don't mess with the `.git` folder.

### 5.1.1 Checking the status of the Git repository

After you initialized a Git repository, you can use `git status` to receive the current file tracking status from Git.

```
git status
```

You should see output similar to the following:

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Let's unpack what this output means:

- `On branch main` tell you that you are currently working on the "main" branch of your Git repository. A branch is like a separate line of development in your project. The default branch is often configured during the setup of Git.
- `No commits yet` means that there haven't been any changes or updates made to the repository yet. You haven't created or saved any new versions of your project.
- `nothing to commit (create/copy files and use "git add" to track)` indicates that there are no new changes to save or "commit" to the repository at this moment. Git is suggesting that if you want to start tracking changes, you should create or copy some files into the repository's directory and then use the `git add` command to tell Git to start keeping track of those files. Once you've added files with `git add`, you can then commit those changes to save them in your project's history.

As you can see `git status` gives you an overview, which files Git is tracking for you. The `git status` command is helpful for understanding the current state of your Git repository and we recommend to use it frequently, especially before adding or committing changes in files to the Git history (see below for more details).

> **Common `git status` flags**
>
> `-s`: Provides a more compact and simplified output, showing only the file status in a short format.
> `-b`: Includes information about the branch you are currently on along with the status.
> `-u`: Shows untracked files in the output.
> `-uno`: Suppresses the display of untracked files.
> `-v`: Provides more detailed information, including additional status information about ignored files.
> You can use these flags in combination. For example, `git status -s -b` will display a short and concise output along with the branch information.

### 5.1.2 Adding files to the Git repository

In the next step, please create an empty text file within your repository, for example using the `touch` command:

```
touch filename.txt
```

You can also use any other method or application for creating files.

## 5.2 Status, staging and committing

### 5.2.1 Checking the status of your Git repository again

After you initialized a Git repository and added a file, you can use `git status` to receive the current file tracking status from Git again. This time, you should see output similar to the following:

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

  filename.txt

nothing added to commit but untracked files present (use "git add" to track)
```

### 5.2.2 Introduction to staging and committing

After initializing a folder with Git, it's capable of recording changes in your files, but it won't do so automatically. In Git, the process of saving changes involves two steps: the staging area and commits. The **staging area** is like a preparation area where you gather and organize your changes before saving them. You can choose which changes to include in a snapshot by adding them to the staging area using `git add`. Once you are happy with the changes in the staging area, you create a commit with `git commit`. A **commit** is like taking a snapshot of your project at a specific moment, capturing all changes in the staging area and saving them as a new version of your project. Commits serve as milestones to track your project's progress and allow you to revert to previous versions if needed. As further introduction to the basic workflow of Git you can check out the last two minutes of the Youtube video: Version control for reproducible research from BERD Academy, which explains the process very vividly.

### 5.2.3 Staging

You can use `git add` to place new or modified files in the staging area. The staging area acts as a space for gathering changes you plan to commit shortly. It bridges between your modified files and the next commit.

For example, you can stage a specific file like this:

```
git add filename.txt
```

If you use `git status` again, your file(s) should now show up under “changes to be committed”.

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   filename.txt
```

Your change to the file (or the addition of it to the project directory) is now **staged**. You do not have to add every file one by one, you can simply use the `-A` flag. This means if you want to stage every file in your project for preparation of your next commit, you can use:

```
git add -A
```

> ⚠ Be careful when staging *all* files!
>
> Especially for existing project folders that have a lot of files in them, be careful when you stage and / or commit all files in the project directory. Git might start to track files that you are actually not interested in tracking. We recommend to use `git status` frequently to check which files are in the staging area and will therefore be added to the next commit.

> 💡 Common `git add` flags
>
> `-A`: Adds all changes, including modifications, deletions, and new files, in the entire working tree.
> `-u`: Adds modifications and deletions, but not new files.
> `--ignore-errors`: Ignores errors when adding files, allowing the command to continue even if some files cannot be added.

### 5.2.4 Committing

Now that the changes to your file(s) are staged, you are ready to **create a commit**.

In simple terms, **a “commit” in Git is like taking a snapshot of your project** at a specific moment in time. It’s a way to save the changes you’ve made to your files. When you make a commit, you’re

saying, “I want to remember what my project looks like right now”. Each commit in Git includes a record of the changes you’ve made, a description of what you did, and a unique identifier. Commits are like milestones in your project’s history, and they allow you to keep track of all the different versions of your work over time. You can go back to any commit to see what your project looked like at that point or even undo changes if needed. Commits help you manage and document the history of your project.

To create a commit, use the `git commit` command followed by the flag `-m` and a commit message **in quotes** that describes the changes you made. The commit message should be short yet informative, providing enough detail to understand the purpose of the commit. If you just use `git commit` without adding a commit massage, the editor of your choosing, opens up and lets you type in a commit message.

```
git commit -m "Add filename.txt file"
```

You should see output similar to the following:

```
[main (root-commit) e9ea807] Add filename.txt file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 filename.txt
```

Congratulations! You have successfully created your first commit in the Git repository. Commits are the core elements of version control and the “commit history” of your Git repository. They allow you to track the history of your project and easily revert changes if needed.

You can use the same workflow of `git add` and `git commit` for every file you add or make changes in.

#### 5.2.4.1 Commit description

Though it’s not required, it can be a good idea to add a more thorough description to the commit, in addition to the (shorter) commit message. While the commit message is usually a single short (less than 50 character) line summarizing the change, a more through description can be used to add more background information or helpful links that may help to understand the changes of the commit.

To add an additional description directly in the command line, add an extra `-m` after the commit message, followed by the description in quotes:

```
git commit -m "Title" -m "Description"
```

> **Common `git commit` flags**
>
> `-m` : Specifies the commit message inline. For example, `git commit -m "Fix typo"` allows you to provide a short commit message directly in the command.
> `-a` or `--all`: Automatically stages all modified and deleted files before committing. This skips the separate `git add` step.
> `-v` or `--verbose`: Provides a detailed output, showing the diff of the changes being committed.

`-e` or `--edit`: Opens the commit message editor, allowing you to edit the commit message before finalizing the commit.
`--amend`: Modifies the previous commit. It allows you to add new changes to the previous commit or modify its commit message.

### 5.2.5 Logging commits

To look at your past commits you can use the `git log` command.

```
git log
```

You should see output similar to the following:

```
commit e9ea80781ceed7cc3d6bff0c7bfa71f320ec1f60 (HEAD -> main)
Author: Jane Doe <jane@example.com>
Date:   Thu Jun 29 12:23:53 2023 +0200

    Add filename.txt file
```

`git log` is useful because it provides a clear and organized view of a repository's commit history. It allows you to track the evolution of a project over time by displaying detailed information about each commit, including changes made, authors, and timestamps. The command can help identifying the source of bugs or issues by pinpointing the commit that introduced them.

Common `git log` flags

`--oneline`: Provides a condensed output with each commit displayed on a single line, showing the abbreviated commit hash and commit message.
`-n` or `--max-count=`: Limits the number of commits shown to the specified . For example, git log -n 5 will display the latest 5 commits.
`--since=`: Shows commits made after the specified . You can use various date formats, such as specific dates or relative expressions like "2 weeks ago" or "yesterday".
`--until=`: Shows commits made before the specified .
`--author=`: Filters commits by the author's name or email using a specified .

### 5.2.6 Comparing versions

Another very handy feature is the `git diff` command. It allows you to **compare two different versions of your file(s)**. By default it shows you any uncommitted changes since the last commit. You can explore this by pasting text in your `.txt` file, for example:

```
Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commo
Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim
```

You can then look at the changes using:

```
git diff
```

You should see output similar to the following:

```
+++ b/filename.txt
@@ -0,0 +1 @@
+Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labo
\ No newline at end of file
```

The output of `git diff` includes several pieces of information:

- `+++` and `---` indicate the paths to the files being compared.
- `@@ -0,0 +1 @@` is a unified diff header. It shows where the changes occurred. Note, that this line will look different for each change or commit.
- `+` indicates added lines and `-` indicates removed lines. Lines that are identical between the two versions are not explicitly shown.

You can also use `git diff` to **compare two specific commits**. For example, if you want to compare the current state of your file with a commit from a few commits ago:

```
git diff <commit-A> <commit-B>
```

Instead of `<commit-A>` and `<commit-B>`, you use the commit hashes, that you can see when using `git log`.

If you've staged some changes using `git add`, you can **compare the staged changes to your last commit**.

```
git diff --staged
```

If you only want to look at the changes of a specific file, you can specify that file.

```
git diff <filename>
```

This command can also be combined with additional flags and options.

> Common `git diff` flags
>
> `--cached` or `--staged`: Shows the changes that are staged (added) but not yet committed.
> `<commit>`: Displays the difference between the current working directory and a specific commit. For example, `git diff abc123` shows the changes compared to commit `abc123`.
> `<commit> .. <commit>`: Shows the difference between two specific commits. For example, `git diff abc123 def456` displays the changes between commit `abc123` and commit `def456`.
> `--name-only`: Outputs only the names of the files that have differences, without showing the actual changes.

`--name-status`: Displays the names of files along with a status identifier that indicates if a file was added, modified, or deleted.
`--color-words`: Highlights the differences at the word level, providing more granular detail.

### 5.2.7 Good commits

You should not consider a commit as a general `Save` button, that you use to save all your recent changes in one go. Instead, each commit should ideally contain one isolated and complete change. For example, if you want to rename a variable and add a new enhancement, put the variable rename in one commit and the enhancement in another commit.

This approach to commits, helps you down the line. For example, if you want to keep the enhancement but revert the renaming of the variable, you can revert the specific commit that contained the variable rename. If you put the variable rename and the enhancement in the same commit or spread the variable rename across multiple commits, you would spend more effort reverting your changes.

### 5.2.8 Commit messages etiquette

Git allows for a title or commit message with a 72 character limit and a description without such a limit. In general you should aim to write clear and short commit messages. There are different conventions across projects/persons but also some general guidelines, you can stick to.

It is standard to start the title with an **imperative verb** to indicate the purpose of the commit, for examople, “add”, “fix” or “improve”. It is also recommended to only write 50 characters for the title and up to 72 for the description (if such a description is even necessary). Here is an example:

```
git commit -m "Implement user registration feature

This commit adds the user registration functionality to the application. It includes the follo
- Created a new 'register' route and view for user registration.
- Added form validation for user input."
```

Commit messages examples

Some examples for commit messages, which stick to the etiquette:

- "Add 'favourites.txt' to the project"
- "Fix typo in first recipe"
- "Improve code comments for clarity"
- "Fix critical security vulnerability (CVE-2022-1234)"
- "Refactor database query functions for efficiency"
- "Update installation instructions in README"

### 5.2.9 Amending a commit

The `git commit --amend` command is used to modify (or "amend") your most recent Git commit in your repository. It allows you to edit the commit message, add more changes to the commit, or simply adjust the last commit without creating a new commit. This flag can be useful if you forgot to include something in your last commit, made a typo or want to change your commit message. You can also combine `--amend` with other flags. To change the commit message of your last commit (without opening an editor) try:

```
git commit --amend -m "changed commit message"
```

Replace `"changed commit message"` with your actual improved commit message.

If you want to to keep your commit message but include new changes to your file(s) in you last commit, you can use:

```
git commit --amend --no-edit
```

The `--no-edit` flag, lets you keep the commit message of your most recent commit.

#### 5.2.9.1 The repeated `--amend` workflow

The "repeated amend" is a Git workflow where you avoid cluttering your history with numerous tiny commits. Instead, you gradually build up a "good" commit by amending it repeatedly. Continue making small changes and amending the existing commit, refining it over time. This method keeps your Git history useful and looking good, making sure your commits tell a straightforward story of how your project evolved.

This workflow can be useful when you're working on a new feature or a significant change and you might make several small adjustments to get it right. Instead of cluttering your history with numerous tiny commits for each tweak, you can use the "repeated amend" to gradually build up a well-polished commit that includes the entire feature.

## 5.3 What files can/should I track with Git?

In principle, any file can be tracked with Git. However, to make use of the full potential of Git, it is recommended to mainly track plain text files with Git.

### 5.3.1 Code files

Tracking code files is the most common and original use case for version control with Git. Git is well-suited for tracking changes in source code, and it's widely used by developers and teams for this purpose. Whether it's a single developer maintaining a personal project or a large team collaborating on a complex software system, Git excels at tracking code files throughout the development lifecycle.

### 5.3.2 Plain textfiles

Git is a useful tool even if your project contains few or no code at all, especially in comparison to a setup that uses emails or shared Dropbox folders for "version control" (see the introduction chapter). However, to really be able to use the full set of features of Git, you should rely on plain text files for your project. This is because `.docx` files are saved as binary files, which makes meaningful outputs of the text inside it impossible for Git. "Plain textfiles" does not mean you have to use `.txt` files. Instead you can use formattable Markdown (`.md`) files. Markdown (`.md`) files are plain text files that contain formatting elements, making them more versatile than plain .txt files. Markdown allows you to add simple formatting like headings, lists, links, and images.

### 5.3.3 Microsoft office files

As said earlier, it is not recommended to track Microsoft office files using Git, since Git treats `.docx` files as binary files. Binary files lack the inherent structure that allows Git to capture and display changes effectively. Git relies on understanding the differences between versions of files. With binary files, you won't benefit from the ability to view detailed textual differences (`git diff`) or effectively use branches. Collaborative work on binary files, such as simultaneous editing of a Word document, may result in complex merge conflicts that are challenging to resolve.

So while it is possible to track `.docx` files with Git, you will not be able to use the many features of Git, which rely on Git being able to display the file, since Git can only output the "zeros and ones" and not the text inside the `.docx` files.

If you ever choose to track them regardless, you should use detailed commit messages, since you will not be able to easily look at the text content of past versions. You will also need to know about temporary word files. For it's own version control system, Word creates temporary files in the folder of your word file. You should not track these files using Git, since Word creates a lot of them. To not stage or commit these files by accident, you can use the `.gitignore` file.

## 5.4 Ignoring files and folders: `.gitignore`

`.gitignore` is a special file. It is used in Git to specify files and directories that should be ignored and not tracked. When you create a `.gitignore` file, place it inside your Git repository and specify filenames inside it. Git will then exclude these files and directories from being staged or committed, e.g. when using `git add -A`. which normally stages all your files.

The `.gitignore` file is useful to prevent certain files or directories that are not essential for the project or generated during the development process from being included in the version history. Files that can recreated from the code, like for example `.png` plots from R code, should also be included in your project's `.gitignore` file. Including only essential files in version control keeps your repository clean, making it easier for collaborators to focus on the important project fules without distractions from unnecessary files. `.gitignore` can also help you avoid accidentally committing sensitive information like passwords, API keys, or personal data, which can lead to serious privacy breaches. Huge files, like big data files should also not be committed, since they can slow down working with your repository. To version control files with a big size, toolsl like DataLad are more appropriate.

To create a `.gitignore` file from the command line, navigate to your repository and use the `touch` command. Alternatively, create the file in your favorite text editor.

```
touch .gitignore
```

`.gitignore` will be a hidden file, to make it show up in the terminal, you will have to use the `-a` flag in the `ls` command. For details on listing of files and folders, see the chapter on the command line.

```
ls -a
```

Now you can write a filename or folder name inside it, to prevent Git from tracking it.

### 5.4.1 Global `.gitignore` file

Instead of specifying files you want Git to ignore for each folder, you can also create global `.gitignore` file. To do this, you create a file named `.gitignore_global` (or any name you prefer) in your users's home directory (located, for example, at `/Users/yourusername`). Then, you configure Git to use this global file by running:

```
git config --global core.excludesfile ~/.gitignore_global
```

After this setup, you can add common files to this list, that you want to ignore. These rules will apply across to *all* Git repositories on your computer.

> **Common wildcards**
>
> As discussed in the command line chapter wildcards are special characters that represent patterns of filenames or directory names. They can be used to specify multiple files or directories that should be ignored by Git when tracking changes. Wildcards allow you to match multiple files with a single rule, making it more convenient to exclude specific types of files or patterns.
> `*`: Matches any number of characters within a filename. For example, *.txt will match all files with the extension .txt in any directory.
> `?`: Matches a single character within a filename. For example, image?.png will match files like image1.png or imageA.png.
> `/`: Matches the root directory of the repository. For example, /config will match a directory named config only in the root of the repository.
> `[]` (Square Brackets): Matches any single character within the brackets. For example, file[123].txt will match file1.txt, file2.txt, or file3.txt.
> `!`: Negates a pattern and includes files that would otherwise be ignored. For example, !important.txt will exclude important.txt from being ignored, even if there's a wildcard pattern that matches it.

Common files to ignore in scientific settings

- **Temporary files and output:** Ignore files generated during analysis, like log files, temporary files, or intermediate data files.
- **Data folders:** Exclude large datasets or data stored locally.
- **Environment-specific files:** Exclude environment-specific files like `.env` files or `venv` folders used for local development setups.
- **System-specific files:** In macOS, ignore `.DS_Store` files, and in Windows, ignore `Thumbs.db` files.
- **R:** Ignore `.Rdata` files and `/Rplots.pdf` generated by R for plotting.
- **Python:** Ignore `.pyc` (Python compiled) files and `pycache` folders.
- **LaTeX:** Ignore auxiliary files like `.aux`, `.log`, `.bbl`, and `.blg` files generated during LaTeX compilation.

Example `.gitignore` file for an R project

```
# R-specific
*.Rproj.user/
*.Rhistory
.RData
.Rproj

# R package specific
.Rcheck/
man/*.Rd
NAMESPACE

# Temporary files
*.bak
*.csv~
*.html
*.pdf

# RMarkdown-specific
*.knit.md
*_cache/
*_files/

# R Markdown Notebook
*.nb.html

# RStudio Project Files
*.Rproj

# R Environment Variables
.Renviron

# R dcf file
DESCRIPTION.meta

# Mac-specific
.DS_Store
```

## 5.5 Bonus exercises

Besides generally playing around with staging, committing, `git log` and `git diff`, you can also do these tasks:

- Add a recipe to your project using the repeated `--amend` workflow

- Initialize a new Git repository and create a `.docx` file in it. Check out the results of `git diff` after writing in the file.
- Use `ls -a` to see the temporary files.
- Create a `.gitignore` file, where you put all temporary files Word creates.

## 5.6 Cheatsheet

| Command | Description |
|---|---|
| `git init` | Initialize folder as Git Repository |
| `git status` | View Git tracking status of files in Repository |
| `git add` | Add file to staging area |
| `git commit` | Commit staged files |
| `git log` | View past commits |
| `git diff` | View made changes compared to the last commit |

## 5.7 Acknowledgements & further reading

We would like to express our gratitude to the following resources, which have been essential in shaping this chapter. We recommend these references for further reading:

| Authors | Title | Website | License | Source |
|---|---|---|---|---|
| The Turing Way Community [17] | The Turing Way: A handbook for reproducible, ethical and collaborative research | 🖵 | License: The process documents and data are made available under a CC BY 4.0 license. Software are made available under an MIT license. Website: https://the-turing-way.netlify.app/ | |
| Millman et al. [14] | Teaching Computational Reproducibility for Neuroimaging | 🖵 | CC BY 4.0. Website: http://dx.doi.org/10.3389/fnins.2018.00727 | |
| McBain [12] | Git for Scientists | 🖵 | CC BY-SA 4.0 | |
| Bryan [5] | Happy Git and GitHub for the useR | 🖵 | CC BY-NC 4.0 | |

# 6 Branches

In this chapter you will learn about the power of branches. Create, manage and merge branches like a pro!

Take the quiz! UHH WS 23/24

Learning Objectives
- ☐ Knowing purpose and benefits of using branches in Git
- ☐ Creating and switching between branches
- ☐ Merging branches and resolving conflicts

Exercises
☐ Create a new branch called `feature/newrecipe` ☐ Add a new recipe to your recipe text file ☐ Merge this branch with your `main` branch and delete the `feature` branch afterwards

## 6.1 Why branches?

Git branches are like separate workspaces within a Git repository. They let you work on different stuff without messing up the main code. You can visualize branches as shown in Figure 6.1.

Each branch keeps its own history, so you can develop things in parallel and easily merge changes between branches when you're ready. It's great for trying out new features, fixing bugs, or experimenting without causing chaos in the main development branch. Scientists might use branches to collaborate on research papers, exploring different parameters while running experiments or to test different data processing techniques or visualization approaches during data analysis. Git makes it easy to switch between branches and handle conflicts when merging. With branches, you can work independently and keep your code organized.

## 6.2 Checking your branches

By default, your work is on the initial branch, usually called `main` (or `master`). We recommend changing your initial branch name to `main`, as discussed in the setup chapter. You can list the available branches in your repository by using `git branch`.

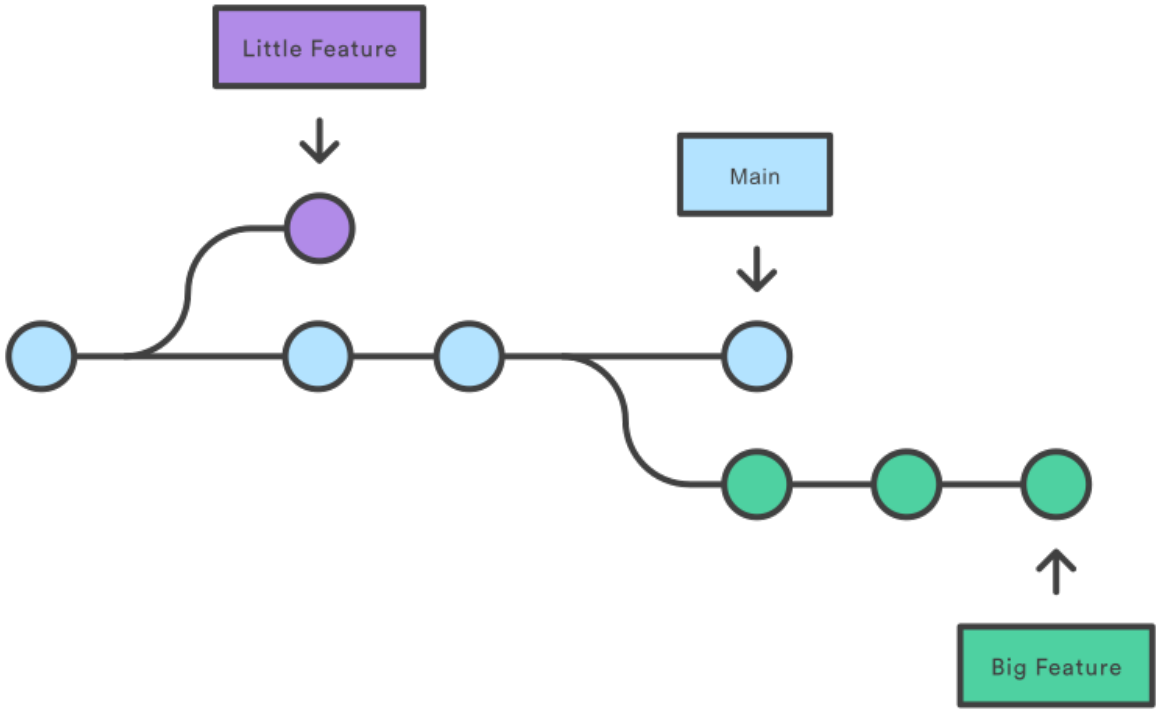



Figure 6.1: Image from Chapter “Git Branch” of the Atlassian “Become a git guru” tutorials, used under a Creative Commons Attribution 2.5 Australia License (direct link)

```
git branch
```

This should return the following output:

```
* main
```

This output indicates that you are currently on the branch named `main`. The asterisk (`*`) in front of the branch name indicates the active or current branch in your Git repository.

In Git, you can have multiple branches in your project, and this command helps you see the list of available branches. The one with the asterisk is the branch where your working directory is currently positioned. In this case, `main` is the active branch. If there were other branches listed without an asterisk, it means they exist, but you are not currently “on” those branches. You are “on `main`”.

## 6.3 Creating a new branch

To create a new branch, use `git branch` followed by a branch name. For example, to create a new branch called `feature` run the following command:

```
git branch feature
```

Now you can use `git branch` again to confirm the creation of the new branch.

```
git branch
```

This should return the following output:

```
  feature
* main
```

Your newly created branch will show up, but your active branch should still be `main`.

> **Common `git branch` flags**
>
> `-a` or `--all`: Lists both local and remote branches.
> `-r or --remote`: Lists only remote branches.
> `-d` or `--delete`: Deletes a specified branch. For example, `git branch -d feature`.
> `-D` or `--force`: Force deletes a branch, even if it has unmerged changes.
> `-m` or `--move`: Renames a branch. For example, `git branch -m old-branch new-branch`.
> `-c` or `--copy`: Creates a new branch by copying an existing branch. For example, `git branch -c existing-branch new-branch`
> `-v` or `--verbose`: Shows more information when listing branches, including the last commit message.
> `--contains`: Filters branches that contain a specific commit.

## 6.4 Switching branches

To switch to another branch, you can use `git checkout` or `git switch`.

For example, enter the following command to switch to the newly created `feature` branch:

```
git switch feature
```

If you switched branches successfully, you should receive an output similar to this:

```
Switched to branch 'feature'
```

Now you can work on this branch as usual using the staging/committing workflow introduced in the chapter on basic Git commands.

> **What is the difference between `git checkout` and `git switch`?**
>
> Prior to Git version 2.23, the `git checkout` command was the primary way to switch branches. It allows you to move to a different branch and update your working directory to reflect the state of that branch. In Git 2.23 and later versions, the `git switch` command was introduced as a safer and more intuitive alternative to `git checkout`. It simplifies the branch-switching process and provides clearer feedback in case of errors.

> `git checkout` is a versatile and powerful command that, among other things, was traditionally used for switching branches. However, it has other use cases like checking out specific files or commits. It can be used for branch switching, but it may have some ambiguity in its syntax, especially when used for other purposes.

### 6.4.1 Error when switching branches

Git is designed to prevent you from switching branches if there's a risk of overwriting your changes. This means that Git will not allow you to switch branches if you have uncommitted changes on your branch. If you try to switch branches with uncommitted changes you will get an error message similar to:

```
error: Your local changes to the following files would be overwritten by checkout:
        chapters/branches.qmd
Please commit your changes or stash them before you switch branches.
Aborting
```

The message specifies the file with modifications, and indicates an aborted branch switch due to uncommitted changes.

If you want to keep your changes you should save them using the stage and commit workflow It is also possible to temporarily stash your changes using `git stash` or delete your changes using `git reset` which is covered in the Stashing and co. chapter.

## 6.5 Merging branches

After you worked on a branch for some time you might want to incorporate your work into the `main` branch. You can do this using the `git merge` command. Switch to your `main` branch (using `git switch` or `git checkout`) and then use:

```
git merge feature
```

Note, that your current branch (the branch that you are "on") should be the one that you want to merge changes *into* from another branch. For example, if you want to merge changes *into* the `main` branch *from* the `feature` branch, you need to be on `main` and then merge the `feature` branch.

If you have not yet made any changes you should get the following output:

```
Already up to date.
```

This output indicates that the branch you are trying to merge is already fully incorporated into the branch you are currently on. In other words, there are no new changes in the `feature` branch that need to be merged because your current branch already contains all the changes from the `feature` branch.

If you have changed file(s) in your folder, you should get an output similar to:

```
Updating 555ba0c..994bb8d
Fast-forward

 example.txt    |   4 ++++

 1 file changed
```

The output reveals a **"fast-forward" merge** operation, where changes from commit `555ba0c` to `994bb8d` are incorporated. The common starting point of the two branches is commit `555ba0c` and commit `994bb8d` marks the latest commit in the merged branch. A "fast-forward" merge, the simplest form of merges, smoothly updates your position to the latest changes without creating a new commit. This linear integration avoids diverging paths. Additionally, the merge modified only one file, the `example.txt` file, introducing four new lines denoted by `4 ++++`.

## 6.6 Merge conflicts

Merge conflicts occur when Git is unable to automatically resolve differences between two branches during a merge. This happens when both branches include conflicting changes to the same part of a file. Git simply can't determine which version of the code should take precedence, and as a result, it requests manual intervention from the developer. It's like when two people try to edit the same part of a file at the same time, and Git gets confused about whose changes to keep. So, it raises its hand and asks you, the developer, to step in and help.

### 6.6.1 Avoiding merge conflicts

Merge conflicts can be a bit of a headache. If you only work locally on your computer, without a remote repository (for example, an online version of your repository on GitHub), you can avoid merge conflicts easier, and might not even need different branches. However, once you start to collaborate with others, it may not be possible to entirely avoid merge conflicts. That being said,

Employing good development practices can minimize the occurrence of merge conflicts. Regularly pulling changes from the remote repository into your local branch (using `git pull`) and keeping your branch up-to-date can help identify and resolve conflicts early. You will learn more about how to effectively manage collaborative work on Git repositories in the following chapters on GitHub.

### 6.6.2 Resolving merge conflicts

When a merge conflict arises, you should not view it as a limitation of Git but rather as a helpful feature. Git is like a referee saying, "Hey, programmers, sort out your differences!"

If you try to merge with conflicting changes in a file the output looks similar to this:

```
Auto-merging example.txt
CONFLICT (content): Merge conflict in example.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Git will highlight the conflicting parts in the affected file(s), and you must choose the desired changes or modify them to create a consistent version. To do this, open up the file(s) with conflicts in a text editor, and you'll see the conflicting sections marked with <<<<<<<, =======, and >>>>>>>.

This might look something like this:

```
\<<<<<<< HEAD
# Recipe: Chocolate Cake

Ingredients:
- 2 cups all-purpose flour
- 1 3/4 cups granulated sugar
- 3/4 cup unsweetened cocoa powder
- ...

Instructions:
1. Preheat the oven to 350°F (175°C).
2. In a large bowl, whisk together dry ingredients.
3. Add eggs, milk, oil, and vanilla. Beat well.
4. Stir in boiling water.
5. Pour into prepared pans.
\=======
# Recipe: Chocolate Brownies

Ingredients:
- 1 cup unsalted butter
- 2 cups granulated sugar
- 4 large eggs
- ...

Instructions:
1. Preheat the oven to 350°F (175°C).
2. Melt butter, then stir in sugar, eggs, and vanilla.
3. Combine dry ingredients and gradually add to the butter mixture.
4. If desired, fold in chopped nuts.
5. Spread the batter evenly into the prepared pan.
\>>>>>>>
```

The lines between <<<<<<< HEAD and ======= represent the changes that were made in your current branch. The section between ======= and >>>>>>> represents the changes from the branch you are trying to merge into your current branch.

Manually edit the conflicting sections to the desired state, removing the conflict markers. You might want to keep some changes from your current branch, some from the incoming branch, or make entirely new changes.

After editing the file it might look something like this:

```
# Recipe: Chocolate Cake

Ingredients:
- 2 cups all-purpose flour
- 1 3/4 cups granulated sugar
- 3/4 cup unsweetened cocoa powder
- ...

Instructions:
1. Preheat the oven to 350°F (175°C).
2. In a large bowl, whisk together dry ingredients.
3. Add eggs, milk, oil, and vanilla. Beat well.
4. Stir in boiling water.
5. Pour into prepared pans.

# Recipe: Chocolate Brownies

Ingredients:
- 1 cup unsalted butter
- 2 cups granulated sugar
- 4 large eggs
- ...

Instructions:
1. Preheat the oven to 350°F (175°C).
2. Melt butter, then stir in sugar, eggs, and vanilla.
3. Combine dry ingredients and gradually add to the butter mixture.
4. If desired, fold in chopped nuts.
5. Spread the batter evenly into the prepared pan.
```

In this example, the conflict has been resolved by keeping both sets of changes. The sections for "Chocolate Cake" and "Chocolate Brownies" are now both present in the file.

If you are done, save the file and use `git add` to stage the resolved changes:

```
git add example.txt
```

Once all conflicts are resolved, proceed with the commit, for example:

```
git commit -m "Resolve merge conflict"
```

Your branches should now be merged and your conflict resolved.

> **Common `git merge` flags**
>
> `--abort`: Abort the current merge operation and reset the branch to its pre-merge state.
> `--continue`: Continue the merge process after resolving conflicts.
> `--log`: Include the log message from the commit(s) being merged in the new merge commit.

### 6.6.3 Types of merges

Git supports different types of merges to cater to various situations in software development. Each merge type has its purpose and advantages, allowing developers to choose the most suitable approach based on their specific project needs and workflow.

#### 6.6.3.1 Standard merge

In a standard merge, Git takes each commit from the merged branch and adds them to the history of the base branch with their original time stamps. It creates a new "merge commit", a special record that marks when the merge happened, showing that the branches are now combined. To perform a standard merge, use the following command:

```
git merge
```

#### 6.6.3.2 Fast Forward Merge

If no new commits were made to the base branch since the feature was created, Git automatically does a "Fast Forward Merge". It's like the standard merge, but no new merge commit is created. It's as if you made the commits directly on the base branch. Since the base branch remained unchanged, there's no need to mark a separate merge in the history. You can ensure a fast-forward merge by using the `--ff-only` flag:

```
git merge --ff-only
```

If you do not want a fast-forward merge, you can ensure a merge commit with the `--no-ff` flag.

```
git merge --no-ff
```

#### 6.6.3.3 Squash and Merge

When you squash a branch, Git combines all commits of the branch into a single commit. This new commit is then added to the history, but the individual commits that were part of the branch are not retained or preserved separately. Instead, the squashed commit represents the collective changes made in the branch, creating a cleaner and more concise history.

```
git merge --squash
```

> Common `git merge` flags
>
> `--interactive` or `-i`: Start an interactive rebase, allowing you to edit, reorder, or squash commits interactively.
> `--continue`: Continue the rebase after resolving conflicts or editing commits during an interactive rebase.

`--abort`: Abort the current rebase operation and return the branch to its original state before the rebase.
`--skip`: Skip the current commit during an interactive rebase.
`-p` or `--preserve-merges`: Preserve merge commits during the rebase.
`--autosquash`: Automatically squash commits marked with "squash" or "fixup" in their commit message during an interactive rebase.

## 6.7 Deleting Branches

Once a branch has served its purpose and is no longer needed, it can be deleted. Removing branches that are no longer active helps maintain a clean and manageable branch structure. To delete a branch you can use the `git branch` command, followed by a `-d` flag.

```
git branch -d feature
```

This should yield an output similar to:

```
Deleted branch feature (was 3343f36).
```

You can only delete a branch, you are not currently on.

## 6.8 Branches best practices

### 6.8.1 Naming branches

Adopt a **consistent naming convention** for branches that accurately **reflects their purpose**. This can include prefixes like `feature/`, `bug/`, or `hotfix/` followed by a descriptive name. Clear and standardized branch names make it easier to identify the purpose of each branch.

### 6.8.2 Branch Lifecycle

Create branches with a **specific goal or task** in mind, and aim to **keep them short-lived**. Long-lived branches can lead to conflicts and make it harder to merge changes. Once the task is complete or the feature is integrated, consider deleting the branch.

# 7 GitHub

This chapter will introduce you to remote repositories, in particular GitHub.

Take the quiz! UHH WS 23/24

Learning Objectives
- ☐ Connecting Git and Github
- ☐ Setting up a GitHub repository
- ☐ Pulling and pushing changes to / from a Github repository

Exercises
- ☐ Upload your repository to GitHub
  - Create a new repository on GitHub.
  - Set the remote URL to your GitHub repository.
  - Push your changes to GitHub.
- ☐ Clone your repository, push and pull
  - Clone your uploaded repository to a different location on your computer.
  - Make changes and push them to GitHub.
  - Pull the changes into your original repository.
- ☐ Bonus: Create a Pull Request
  - Create a new branch in your local repository.
  - Add, commit, and push changes in the new branch.
  - Create and merge a pull request for this new branch on GitHub

## 7.1 What are remote repositories

A remote repository is a version of your Git repository that is hosted on a server, mostly on the internet. Unlike your local repository, which is on your personal computer, a remote repository can be a shared resource that allows multiple contributors to collaborate on a project. These remote repositories could be hosted on platforms like GitHub, GitLab, Bitbucket, or a private server.

### 7.1.1 Why use remote repositories?

One of the primary reasons to use remote repositories is for **collaboration**. They provide a centralized location where team members can collectively work on a project. Whether contributors are across the room or across the globe, remote repositories make it possible for everyone to access the same set of project files and contribute to the development process.

Hosting a project on a remote repository also serves as a **backup mechanism**. If a contributor's computer fails or their files get corrupted, they can always clone a fresh copy of the project from the remote repository. This ensures that the project's progress is not dependent on a single person's local environment.

## 7.2 What is GitHub?

GitHub is a popular web-based platform that hosts Git repositories remotely. You can use it in your browser without installing any software on your computer. It's a collaborative environment for software development, allowing developers to work together, manage code, and track changes. GitHub also offers project management and code review tools, making it great for open-source development and team collaboration.

## 7.3 Creating a GitHub account

To be able to use GitHub, you will need to create an account:

1. Open https://github.com in your browser
2. Click the `Sign up` button
3. Enter the required personal details
4. In step 2 select the free plan.

**Choosing a Github Name**

Here are some tips for choosing a GitHub username:

- Incorporate your name.
- Adapt your username from other platforms.
- Choose a professional username. Your future boss might look at your GitHub profile.
- Opt for a shorter username.
- Be unique and concise.
- Avoid references to specific institutions.
- Use all lowercase and hyphens for word separation.

**GitHub student education pack**

The GitHub Student Developer Pack is a program offered by GitHub to students, providing them with free access to a collection of valuable developer tools and services. To benefit from

the pack, visit the GitHub Education website and sign up with your student email to verify your academic status. Once verified, you gain access to various resources, including GitHub Pro with unlimited private repositories, free domain names, cloud credits, coding courses, and more.

## 7.4 Connecting to GitHub

### 7.4.1 Authentication

GitHub authentication is needed to access and work with repositories that are stored on GitHub. It is kind of like showing your ID to prove who you are before entering a restricted area. It's necessary to make sure only the right people or programs can access and do things on GitHub, keeping everything safe and organized.

For security reasons it is no longer possible to authenticate yourself using only your GitHub email and password. There are multiple authentication methods which offer different advantages and drawbacks. In this section, we will briefly introduce one of the two most common authentication methods: Personal Access Tokens.

If you would rather use the SSH protocol for authentication, check out the SSH documentation on Github.com.

#### 7.4.1.1 Personal Access Token

Follow these steps to set it up:

1. Log in to your GitHub account on GitHub.com
2. Click on your profile picture in the top-right corner and select "**Settings**".
3. In the left sidebar, select "**Developer settings**", then "**Personal access tokens (classic)**".
4. Click "**Generate new token**".
5. Give your token a **name**. The name should help you to later identify in which context you used the token. For example, you could add a short description of the machine that you generated the token for.
6. Choose the desired **scopes**. The scopes define the permissions of the token, or "what it is allowed to do" with your GitHub account and repositories. For a normal usecase, you should enable the scopes: **repo**, **admin:org** and **delete_repo** for a full access.
7. Set an expiration date if needed. You can also choose "No expiration".

Click "**Generate token**," and you'll be presented with your new token.

Make sure to copy it as it won't be shown again. The first time you try to interact with a remote repository, you will get asked for a password. **Paste this key as a password into the terminal to proceed.**

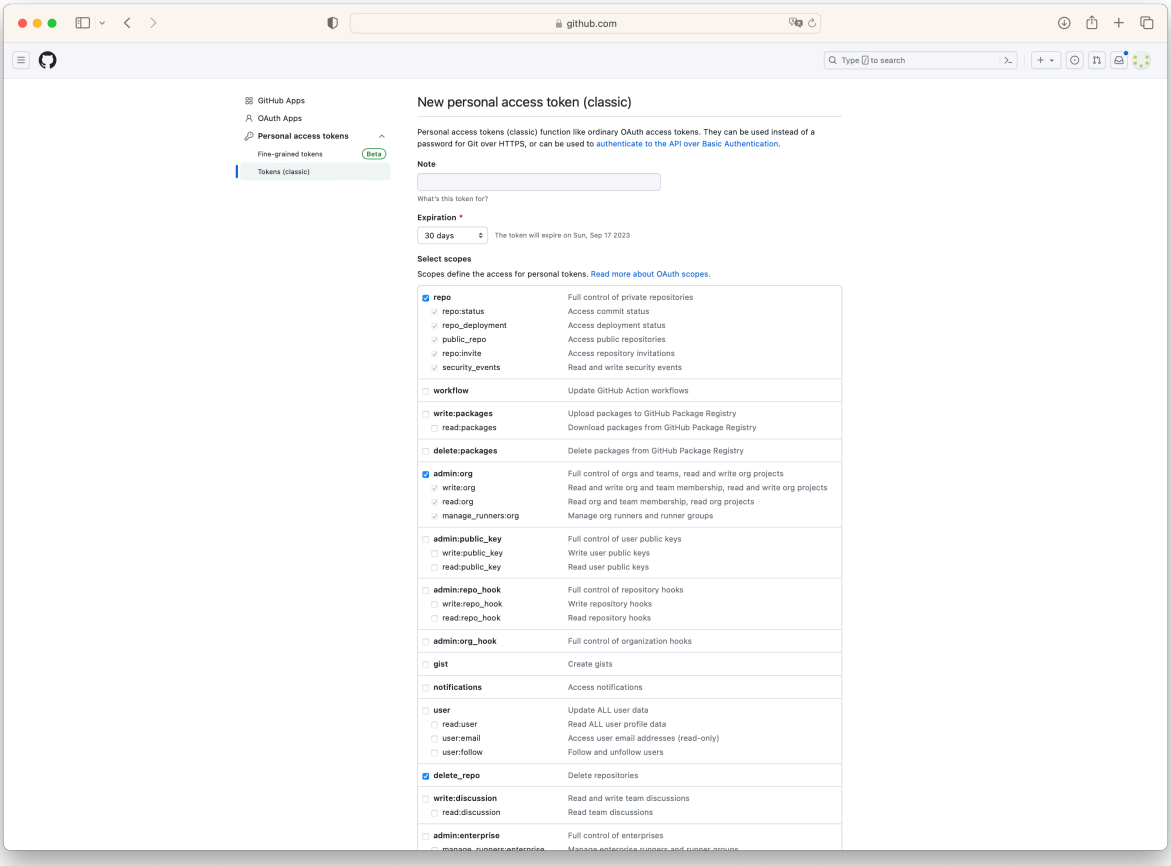



Figure 7.1: Screenshot of the personal access token settings

If you use Git Bash on Windows, your terminal will open a window where you can choose to authenticate yourself with a PAT.

More about PAT settings

**Expiration Date:**
When you create a Personal Access Token, you can specify its expiration date. The expiration date is the date and time when the token becomes invalid and cannot be used for authentication anymore. Setting an expiration date is a security measure to limit the time during which the token can be misused if it falls into the wrong hands. Creating an expiration date for a GitHub Personal Access Token (PAT) is not strictly necessary, but it is considered a good security practice.
**Scopes:**
When you create a PAT, you can choose the scopes to define what actions the token is allowed to perform. These scopes determine the level of access the token has to different parts of your GitHub account or repositories.
Here are some common scopes you might encounter when creating a GitHub PAT:
`repo`: This scope provides full control of private and public repositories, including the ability to read, write, and delete code.
`repo:status`: Grants access to commit statuses, allowing the token to read and set commit status for a repository.
`repo_deployment`: Enables the token to access deployment-related events and perform actions related to repository deployments.
`read:org`: Allows the token to read organization membership, teams, and repositories within an organization.
`user`: Grants access to user-related data, including user profile information.
`delete_repo`: Provides permission to delete repositories.
`admin:org`: Offers administrative access to the entire organization, including managing teams and repositories.

## 7.5 GitHub repositories

### 7.5.1 Creating a GitHub repository

To create a new repository, click on the "**+**" sign in the top-right corner of the GitHub page. From the dropdown menu, select **New repository**." On the new repository page, as shown in Figure 7.2, enter a name for your repository and make sure the "**Public**" option is selected if you want it to be accessible to everyone. You can also choose to create a README file by checking the corresponding box. If you want to upload an existing repository, you should not create a README, since your GitHub repository needs to be completely empty. Finally, click the green **"Create repository"** button at the bottom of the page to complete the creation process.

Public or private?

When you create a new repository on GitHub, you can control who can see it by choosing if it's public or private.

If it's public, anyone on the internet can access it. If it's private, only you, the people you specifically allow, and, for organization repositories, certain members can access it. For further details, see the chapter "About repository visibility" in the GitHub documentation.
If you have admin permissions for a repository, you can change its visibility. This means you can later still make a public repository private and vice verca, if needed.

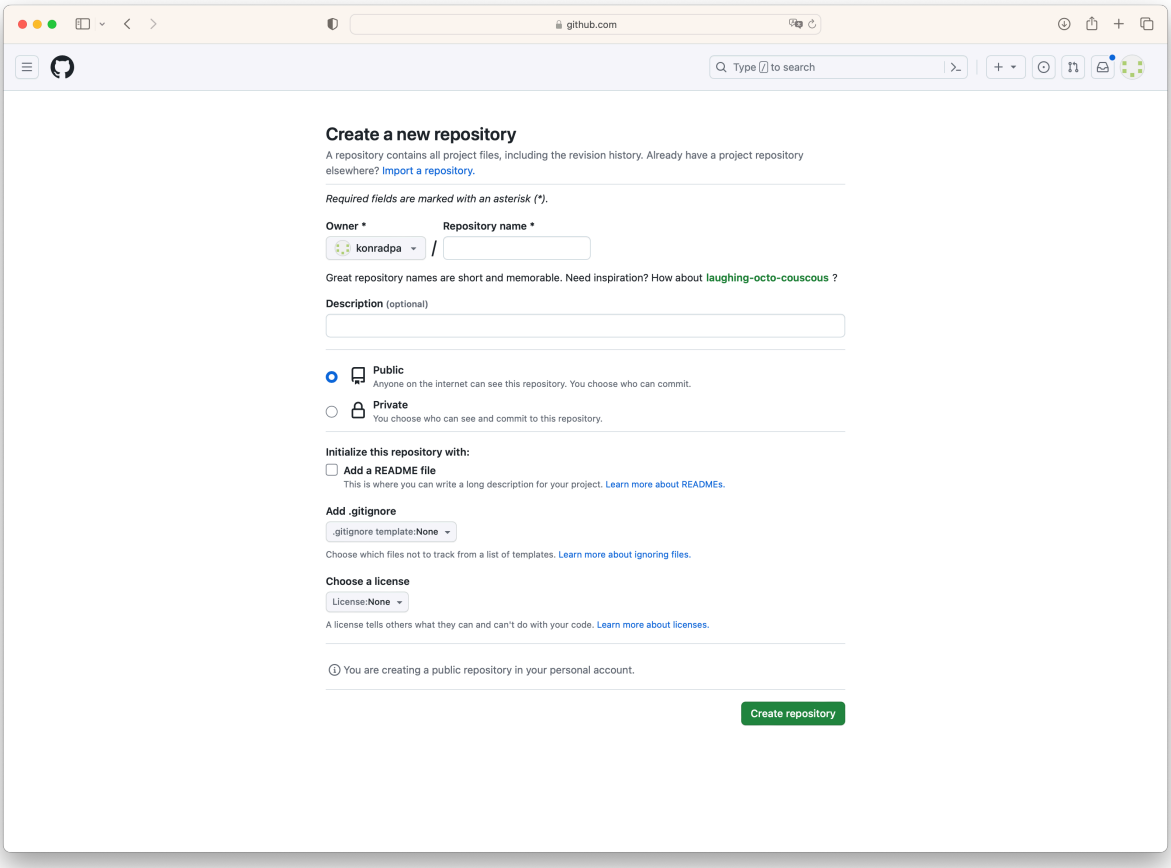



Figure 7.2: Screenshot of creating a new repository

### 7.5.2 Committing changes on GitHub directly

Once you have created the repository and initialized it with a `README.md` file, you will find the `README.md` file in your repository. You can edit this file to provide information and details about your project. Whatever you write in the `README.md` file will be displayed on the front page of your project.

To add more files to your repository, click on the "**Add file**" button and choose whether to upload existing files or create a new file directly on GitHub. You can edit and commit changes to every file in your repository. The commit message helps you keep track of the changes you've made and serves as a brief summary of the modifications made to the repository.

### 7.5.3 Uploading your repository

To initialize an existing folder as a GitHub repository, you first need to create a new repository, like explained above. You will then need to initialize the folder as a Git repository using `git init`. If you have already done this, you can of course skip this.

To add your folder as a remote repository, navigate to your folder using the command line and use the `git remote` command.

```
git remote add origin https://github.com/yourusername/repositoryname.git
```

This command is essentially telling Git: "Add a remote repository named `origin` with the location (URL) `https://github.com/yourusername/repositoryname.git`". After running this command, your local Git repository will be aware of the remote repository, and you can push and pull changes between your local repository and the one on GitHub. The term "origin" is a conventionally used name for the default remote repository. It's a standard name, but you could technically choose another name if you prefer.

Your are now able to push your files to the remote repo using `git push`.

```
git push -u origin main
```

This pushes the local branch named "main" to a remote repository called "origin" while setting it as the upstream branch. `-u` is used to set the upstream branch. It tells Git to remember the remote branch to which your local branch should be pushed in the future.

common git remote commands

`git remote`: Lists all remote repositories associated with the current local repository.
`git remote -v`: Lists remote repositories along with their URLs.
`git remote add <name> <url>`: Adds a new remote repository with the specified name and URL.
`git remote rm <name>`: Removes the remote repository with the specified name.
`git remote rename <old-name> <new-name>`: Renames a remote repository from old-name to new-name.

### 7.5.4 Cloning a Repository

Cloning refers to creating a local copy of a repository from GitHub on your computer. When you clone a repository, you download all of its files, commit history, and branches to your local machine. Cloning allows you to work with the repository locally, make changes, and push those changes back to the original repository if you have write access. It is commonly used when you want to contribute to a project or work on it independently. To clone a repository, you need the repository's URL from GitHub. You can find the repository's HTTPS URL on the repository page, if you click on the green **Code** button. Then use the terminal to navigate to the directory where you want to clone the repository. Then use the `git clone` command followed byh the URL of the remote repository.

```
git clone https://github.com/username/repository.git
```

Once the cloning process is complete, you will have a local copy of the Git repository in the directory you specified. You can now work with the files in the repository, make changes, and commit your modifications locally. Remember to use the appropriate Git commands like `git add`, `git commit`, and `git push` to manage your changes and synchronize them with the remote repository as needed. By default, `git clone` will create a reference to the remote repository called "origin", so you do not have to use `git remote` to create a remote repository.

### 7.5.5 Repository settings

To access the settings of a GitHub repository, go to the main page of the GitHub repository you want to configure and look for the "Settings" tab, usually located towards the right side of the repository's navigation bar. Once you're on the repository settings page, you'll find various sections on the left sidebar that allow you to configure different aspects of the repository, such as "General," "Branches," or "Collaborators".

In the "General" section you can, for example, rename your repository or default branch, change the repositories visibility or delete your repository.

In the "Collaborators" tab, it is possible to add accounts which can view or change your repository, even if it is private. For a detailed guide on the repository settings, you can check out the GitHub documentation.

### 7.5.6 Fetching

The command `git fetch` is used to keep up-to-date with changes in the remote repository, *without* merging them into your local branch. Instead Git will update all your local tracking branches to the remote branch. To view your remote tracking branches, you can use `git branch -r`. After fetching, you can use `git switch` to review the changes on these branches. If you want to integrate the changes into you local branch, you can either now use `git pull` to directly update your local branch, or use `git merge origin/branchname` when you are on your local branch that you want to update.

`git fetch` is the safer option compared to `git pull` since it retrieves the changes from the remote repository without making any changes to your local working directory and staging area.

## 7.6 Pulling

The `git pull` command is used to retrieve the latest changes from a remote repository **and** merge them into your local branch. Effectively `git pull` first runs `git fetch` (see above) to fetch the latest changes from the remote repository and then integrates these changes into the local branch, either using `git merge` (for details, see the branches chapter) or `git rebase` (for details, see the section on rebasing), depending on the configuration. In other words, `git pull` is a `git fetch` followed by a `git merge`. You can test this function by editing a file in your repository on GitHub

and then use `git pull` to update your local file with the change made on GitHub. To edit a file, first click on it, in the repository. In the top-right corner of the file view, click on the pencil icon ("Edit this file") to start editing. Make the necessary changes to the file in the in-browser editor. You can add, modify, or delete content as needed. As you make changes, GitHub automatically tracks your modifications in the editor, displaying them as "changes not staged for commit". Provide a brief description of the changes in the "Commit changes" section at the bottom of the page.

Now you can open up your terminal once again, and in the relevant directory use the `git pull` command to update your local file with the change made on GitHub.

```
git pull
```

If the pull is successful and there are no conflicts, you might see a message like:

```
Updating abc123..def456
Fast-forward file.txt | 2 +- 1 file changed,
1 insertion(+), 1 deletion(-)
```

The first part of the message indicates the range of commits that were fetched and merged. In this example, it suggests that Git is updating from commit `abc123` to commit `def456`. The "Fast-forward" message indicates that the local branch could be updated by moving the branch pointer forward, incorporating the new commits from the remote branch. This is possible when the local branch has not diverged from the remote branch, and there are no local commits that the remote branch does not have. The next line shows changes in the file `file.txt`. The `2 +-` indicates that two lines were changed with one insertion and one deletion. The `+` represents an addition, and the `-` represents a deletion. The last line of the message provides a summary of the changes. It states that one file was changed, with one line inserted and one line deleted.

Be aware which branch you are pulling!

A regular `git pull` incorporates changes from a remote repository into the **current** branch. You can use the `git branch` command to verify which branch you are currently on (for details, see the branches chapter). If the current branch is behind the remote, then by default it will fast-forward the current branch to match the remote. If the current branch and the remote have diverged, the user needs to specify how to reconcile the divergent branches.

Common `git pull` command flags

`--ff-only`: Perform a fast-forward merge only. If the remote branch has new changes, Git will only update your local branch if a fast-forward merge is possible. Otherwise, it will abort the pull.
`--no-commit`: Perform the pull, but do not create an automatic commit after merging. This allows you to review the changes before committing manually.
`--verbose` or `-v`: Provide more detailed output during the pull operation. This can be helpful for understanding the actions Git is performing.
`--squash`: Instead of a regular merge or rebase, squash all the changes into a single commit. This can be useful for cleaning up the commit history or grouping related changes together.
`--autostash`: Automatically stash local changes before pulling, then apply them back after

the pull is complete. This is handy when you have changes in progress and want to pull in the latest changes from the remote branch.

## 7.7 Pushing

The`git push` command is like a reversed `git pull`. It is used to upload your local commits to the remote repository. When you run `git push`, Git examines your local branch and its commits, and then pushes those commits to the corresponding branch on the remote repository. Git will verify if your local branch is up to date with the remote branch. If there are new commits on the remote branch that you don't have locally, Git may reject the push and ask you to first pull the latest changes and merge them into your local branch to prevent overwriting or conflicts. It's important to note that you need appropriate access and permissions to push to a remote repository. If you do not have write access, you won't be able to push your changes. If you do not specify a branch, Git will push the one you are on in the moment.

```
git push origin main
```

This pushes your local commits to the remote repository called `origin` on the branch `main`. After a successful push, you will get a message like this:

```
Enumerating objects: 17, done.
Counting objects: 100% (17/17), done.
Delta compression using up to 8 threads
Compressing objects: 100% (12/12), done.
Writing objects: 100% (12/12), 2.11 KiB | 2.11 MiB/s, done.
Total 12 (delta 9), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (9/9), completed with 5 local objects.
To https://github.com/username/repository.git
   fb3efef..8f50685  main -> main
```

The `git push` operation involves enumerating, compressing, and writing objects to the remote repository. The output indicates the progress of these steps, including delta compression, and concludes with a summary of the pushed branch and commit range. In this specific example, changes from the local main branch (commit `fb3efef`) were pushed to the remote main branch (commit `8f50685`).

Common `git push` flags

`-u` or `--set-upstream`: This flag is used to set the upstream branch for the current branch. It is typically used when pushing a branch for the first time to link the local branch with a remote branch.
`--force` or `-f`: Force pushes the local main branch to the origin remote repository, overwriting any changes that may have been made to the main branch in the remote repository.
`--all`: This flag pushes all branches to the remote repository.
`--dry-run`: This flag simulates the push operation without actually pushing any data to the

remote repository. It's useful for checking what would be pushed.
`local_branch:remote_branch`: This syntax allows you to push a specific local branch to a specific remote branch. Example: `git push origin my_local_branch:my_remote_branch`

## 7.8 Extra features

### 7.8.1 Forking a repository

Forking a repository on GitHub allows you to create a personal copy of a project under your GitHub account.

This separate copy includes all files, commit history, and branches. Forking is commonly used when you want to contribute to a project without directly modifying the original repository. It enables you to make changes independently, create new branches, and push modifications to your forked repository. You can also submit changes to the original repository through pull requests.

By clicking the **Fork** button on a repository page, shown in Figure 7.3, you create an identical copy in your GitHub account.

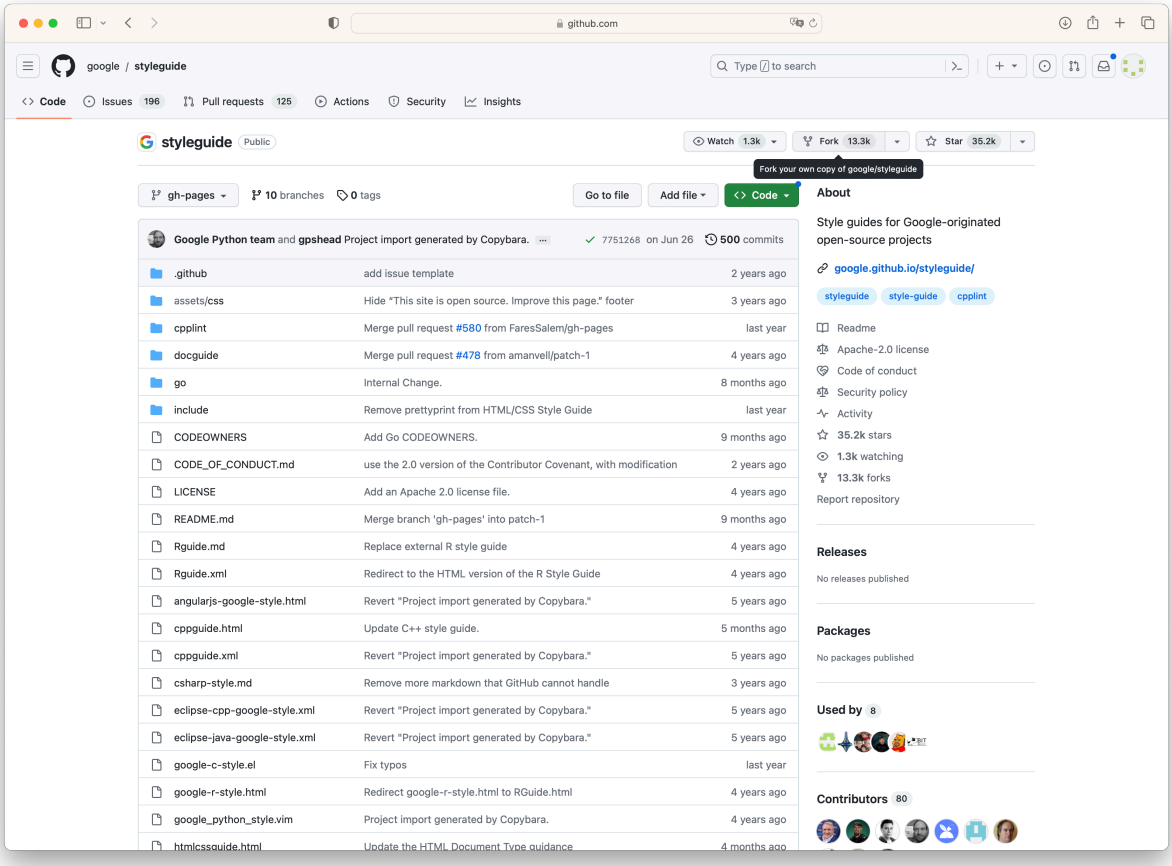



Figure 7.3: Screenshot of a repository with fork button

Once the forking process is complete, you can clone the repository to your local machine using the `git clone` command.

### 7.8.2 Pull requests

A **pull request** serves as a request for code review and integration into a project's codebase, enabling collaboration and ensuring code quality before changes are merged. When dealing with pull requests, there are two main workflows: (1) a pull request from a forked repository, (2) a pull request from a branch within a repository.

#### 7.8.2.1 Pull request from a forked repository

To create a pull request after forking a repository and making changes follow these steps:

1. Navigate to your forked repository on GitHub by visiting https://github.com//. Make sure you are on the branch that contains the changes you want to propose.

2. Click on "Contribute" and then "Open Pull Request".

3. In the pull request interface, choose the branch you made changes to in your fork (the "compare" branch) and the original repository's branch where you want to propose your changes (the "base" branch).

4. Review the changes.

5. Click on the "Create Pull Request" button. Give your pull request a meaningful title and provide a description explaining the changes you made.

6. Submit the Pull Request.

The owner of the original repository can now approve the changes and thereby merge your repository's branch into his.

#### 7.8.2.2 Pull request from a branch within a repository

In this scenario, contributors work directly within the main repository, creating a new branch for their changes. After completing the changes in the branch, they create a pull request from that branch to the main repository's default branch (typically `main` or `master`). The workflow for this is the same as for opening a pull request for a forked repository.

### 7.8.3 `README` file

As mentioned earlier, the `README` file can be created together with your repository or can be added later. The `README` file should contain "a description of your repository" but what does that entail?

The exact content depends on your repository, but some general things that you might want to include are:

- **Project description**: What function does this repository serve, what are it's key features?
- **Installation instructions** (if applicable): Explain how to install and set up your project, including any dependencies or prerequisites. Provide clear instructions to help users or contributors get started with your project quickly.
- **Usage** (if applicable): Provide examples or code snippets demonstrating how to use your project.
- **Contributing**: If you welcome contributions, specify how others can contribute to your project. Here, you can also include guidelines for submitting bug reports, feature requests, or pull requests.

> **Adding a `CONTRIBUTING.md` file to a repository**
>
> For larger or more complex projects where contributions may involve setting up a specific development environment or adhering to specific workflows, it is standard practice to create a file called `CONTRIBUTING.md`. GitHub recognizes the presence of a `CONTRIBUTING.md` file in a repository and, for example, automatically includes a link to the `CONTRIBUTING.md` file when users open a new issue or pull request.

- **Acknowledgments**: Give credit to any third-party libraries, tools, or individuals that contributed to your project.
- **License**: Choose a license that aligns with your project's goals. Common options include GPL, allowing modifications and commercial use. You can use choosealicense.com for guidance. The chosen license influences contributions to your project.

#### Markdown syntax

A `README` is typically a Markdown (`.md`) file, which means you can use Markdown syntax in it. Markdown allows you to easily format text, create lists, include links, and embed images. For an introduction, you can check out the Quarto workshop slides by the creators of this book.

### 7.8.4 Git blame

The command `git blame` is useful when you are working on a project with multiple collaborators. It it used to show when and by whom, each line in a specific file was last changed. You have to specify a file when using it, for example:

```
git blame recipes.txt
```

If you are working solo on a text file, the result should not be very interesting. However if you collaborate on code, this command can be very useful (for blame **or** praise!)

> Common `git blame` flags
>
> `-L <start>,<end>`: Specifies a line range to blame. For example, `git blame -L 10,20 file.txt` will blame lines 10 to 20 of the file.
> `-M`: Detects lines that were moved within the file and shows the original author's information.
> `-e`: Shows the author's email address in addition to their name.
> `-t`: Shows the commit timestamp for each line.

### 7.8.5 Template repositories

Template repositories, or repository templates, on GitHub enable you to create a repository that acts as a starting point for others. Instead of beginning a new project from scratch, you can use a template repository that already includes predefined files, directories, and even code. This simplifies the process of setting up new projects that share common characteristics or follow best practices. Template repositories have a "Use this template" button on a template repository page, as shown in Figure 7.4. Click it to provide a new, repository name and description, and then create the repository with your desired settings.

### 7.8.6 Branch protection

When using Git hosting platforms like GitHub or GitLab, it's a good idea to enable branch protection rules for your critical branch(es), typically your `main` or `master` branch. By protecting it, you ensure that only reviewed and/or tested code gets merged into this branch. It also prevents accidental or unauthorized changes from being merged directly.

#### 7.8.6.1 How to protect the `main` branch on GitHub

- Go to your repository on GitHub and click on "Settings".
- In the left sidebar, select "Branches".
- Under "Branch protection rules," click on "Add rule".
- In the "Branch name pattern" field, enter the name of your `main` branch.
- Enable the options you want to enforce for the `main` branch.
- Click "Create" to save the branch protection rule.

A common option includes to **require pull request reviews**. This requires one or more approving reviews before changes can be merged. Optionally, you can enable other protections, such as preventing force pushes or deleting the branch.

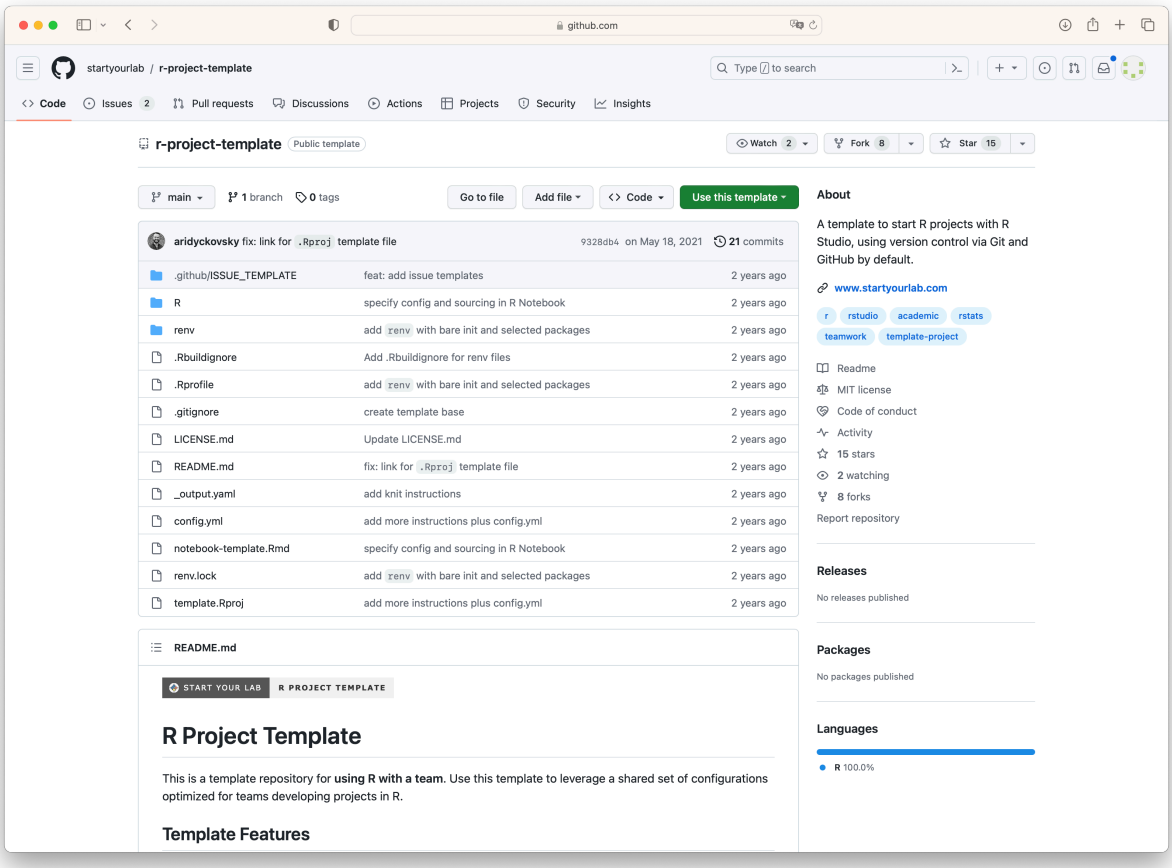



Figure 7.4: Screenshot of a template repository

### 7.8.7 Alternatives to GitHub

While GitHub is one of the most popular and widely used platforms for hosting Git repositories, there many alternatives to GitHub. Not only since it was bought by Microsoft in 2018, who subsequently used public repositories to train their AI tool Some prominent options include GitLab, Bitbucket, SourceForge or Codeberg.

Potential disadvantages of GitHub include limited free private repositories, privacy concerns and a potential vendor lock in. On the other hand there are also advantages to GitHub like it's relatively easy to use interface, the rich ecosystem and it's extensive integration to third party tools.

## 7.9 Acknowledgements

| Authors | Title | Website | License | Source |
|---|---|---|---|---|
| The Turing Way Community [17] | The Turing Way: A handbook for reproducible, ethical and collaborative research | | License: The process documents and data are made available under a CC BY 4.0 license. Software are made available under an MIT license. Website: https://the-turing-way.netlify.app/ | |
| Chacon and Straub [7] | Pro Git | | CC BY-NC | |
| GitHub [9] | GitHub Docs | | CC BY-NC 4.0 | |

## 7.10 Cheatsheet

| Command | Description |
|---|---|
| `git clone` | Create a local copy of a repository |
| `git pull` | Fetches and merges the latest changes from a remote repository into your current branch |
| `git fetch` | Updates your remote tracking branches |
| `git push` | Upload your local commits to a remote repository |
| `git blame` | shows the authorship and commit information of each line in a file |
| `git remote` | manages remote repositories |

# 8 Tags and Releases

Let's dive into the world of Git tags and releases! We'll cover everything from creating tags to managing versions, sharing tips for smooth collaboration and integration with Zenodo.

Learning Objectives
- ☐ Understand why Git tags matter in version control and project management.
- ☐ Learn when to use lightweight or annotated tags and how to apply them.
- ☐ Get the hang of pushing and pulling tags in Git for seamless teamwork.
- ☐ Explore how GitHub releases complement Git tags
- ☐ Discover how to use Zenodo to make your repository citable

Exercises
☐ Create a Lightweight Tag ☐ Create an Annotated Tag ☐ Switch to a Tag ☐ Delete a Tag ☐ Push a Tag to remote ☐ Create a GitHub release ☐ Link a Zenodo record

## 8.1 Tags

### 8.1.1 Importance of Tags

At its core, a tag in version control systems like Git is a reference or a label associated with a **specific commit**. Tags are used to mark particular points in a project's history, usually to signify important **milestones** or **releases**. It acts as a human-readable reference to a particular state of the repository.

Once created, a tag is generally immutable It doesn't move or change its associated commit. Tags don't store changes themselves. They merely point to an existing commit. Tags should be named in a way that is meaningful and easy to understand. Commonly used names include version numbers (for example, `v1.0`, a practice known as semantic versioning), release names, or other identifiers that convey the purpose of the tag.

Tags can also be beneficial for collaboration within a development team, as they provide a **common reference point** for discussing and working on specific versions of the project. Unlike branches, which represent separate lines of development, tags are typically independent of branches. They can be created on any commit, regardless of the branch. Tags are stable references that don't change even if new commits are added to the repository.

There are two types of tags: Lightweight and Annotated tags.

### 8.1.2 Lightweight Tags

Lightweight tags in Git are simple pointers to specific commits in the version control history. Unlike annotated tags, they carry **minimal metadata**, consisting only of the tag name and the hash of the associated commit. Lightweight tags are created using `git tag`, followed by the tag name:

```
git tag v1.0                                                    ①
```

① `v1.0` is only an example. You can choose any other tag name.

This creates a lightweight tag named `v1.0` that points to the current commit at the time of creating the tag. If you create a lightweight tag in this way (without specifying another commit), it is created at the current commit (`HEAD`) in your working directory. If you want to tag another specific commit, you include the commit hash when creating the tag:

```
git tag v1.1 <commit-hash>
```

Replace `commit-hash` with the actual hash of the commit that you want to tag.

You can verify that the tag has been created by listing all the tags. To do this, use:

```
git tag
```

### 8.1.3 Annotated Tags

Annotated tags in Git are tags that include **additional metadata** beyond a reference to a specific commit. When you create an annotated tag, Git stores information such as the tagger's **name**, **email**, the **date** the tag was created, and a **tagging message**. This additional information provides context and details about the tag, making annotated tags more informative compared to lightweight tags. To create an annotated tag, you use `git tag` with the `-a` flag. To include a tagging message, the `-m` flag is used If you do not use the `-m` flag, your text editor of choice will open up. The tagging message can be a detailed description of the tag, such as release notes or significant changes. For example:

```
git tag -a v1.0 -m "Release version 1.0"
```

Annotated tags are larger in size compared to lightweight tags due to the additional metadata.

> Common `git tag` flags
>
> `-a` or `--annotate`: Creates an annotated tag, which includes additional metadata like tagger information and a tagging message.
> `-m` or `--message`: Specifies the tagging message directly in the command. Useful when creating an annotated tag without opening an editor.
> `-l` or `--list`: Lists existing tags.
> `-d` or `--delete`: Deletes a tag.
> `-f` or `--force`: git tag -f v1.0

### 8.1.4 Tags workflow

Once you created a lightweight or annotated tag, it will show up if you use `git tag` or `git tag -l`. To switch to a tag, you can use `git checkout` followed by the name of the tag:

```
git checkout v1.0
```

Now you are "on the tag" and your working directory reflects the state of the code of the commit the tag is pointing to.

In general, it's not advisable to make changes directly after switching to a tag. When you switch to a tag in Git, you enter a "detached `HEAD`" state, meaning that you're not on any branch. In this state, any new commits you make won't belong to any branch, and you might lose those changes if you switch away from the tag. If you need to make changes based on a specific tag, it's often better to create a new branch from the tag and then make your changes on that branch.

A recommended practice is to treat **annotated tags as suitable for public releases** and **lightweight tags as more appropriate for private use**. Annotated tags include additional metadata such as the tagger's name, email, and date, making them valuable for public releases where comprehensive information is beneficial. On the other hand, lightweight tags act as simple "bookmarks" to a commit, serving as efficient pointers without the extra metadata. They are particularly useful for creating quick links to relevant commits in a more private or internal context.

### 8.1.5 Pushing and pulling tags

Pushing and pulling tags in Git involves syncing tags between your local repository and a remote repository, such as GitHub, as discussed in the GitHub chapter. To push a specific tag to the remote repository, use:

```
git push origin <tag-name>
```

Replace `<tag-name>` with the name of the tag you want to push.

To push all tags to the remote repository, use:

```
git push origin --tags
```

To fetch all tags from the remote repository (without merging), use:

```
git fetch --tags
```

Alternatively, you can combine fetching and merging using:

```
git pull --tags
```

When you pull or fetch tags, Git fetches tag references, but it does not automatically switch to the state of a specific tag. If you want to work on a specific tag after pulling, use `git checkout` to switch to that tag.

## 8.2 GitHub Releases

Releases represent specific versions of your repository, that you can package and share with a broader audience for download and usage. These releases are tied to specific Git tags, which act as markers for specific points in your repository's history.

Here's a step-by-step guide:

1. Go to the **main page** of your GitHub **repository**.
2. Click on "**Create a new release**" under "**Releases**" tab located on the right side of your repository page. Now, you should see a page, similar to Figure 8.1.
3. Choose an **existing Git tag** or **create a new one**. If you have created and pushed a Tag, it should show up here. Enter the **tag version** in the "Tag version" field. Enter a **meaningful title** for your release. In the "**Describe this release**" field, provide **release notes**. This can include details about new features, bug fixes, and any other relevant information. If you have binary files, installers, or other assets related to the release, you can attach them by clicking on "Attach binaries by dropping them here or selecting them".
4. If you're creating a draft release, you can save it as a draft by clicking on the "Save draft" button. Draft releases are not visible to the public. If you're ready to make the release public, click on the "**Publish release**" button.

That's it! You've successfully created a GitHub release for your project! □

After publishing the release, you and others can view it on the **Releases page**. It will include the release notes, associated Git tag, and any attached assets.

## 8.3 Zenodo

Zenodo is an open-access digital repository platform designed to **preserve and share research outputs**. It is operated by CERN (European Organization for Nuclear Research) and supported by the European Commission. Zenodo provides a platform for researchers across various disciplines to deposit, share, and archive their scholarly works, datasets, code, and other research outputs.

Zenodo can be integrated with version control systems like Git and platforms like GitHub. This integration allows for **automated archiving of specific releases or tags** from Git repositories. Zenodo assigns a **Digital Object Identifier (DOI)** to each record, including those linked to GitHub repositories, providing a permanent link for citation.

By linking your GitHub repository, you also ensure that your work is archived and accessible beyond the lifespan of GitHub. When you publish a research paper, journals increasingly require or encourage the deposition of associated data in a public repository. Uploading the contents of your GitHub repository to Zenodo ensures that your work is openly accessible and can be cited.

> What is a Digital Object Identifier (DOI)?
>
> A Digital Object Identifier (DOI) is a unique alphanumeric string assigned to a digital document or resource to provide a permanent and stable link to it. DOIs are commonly used to identify

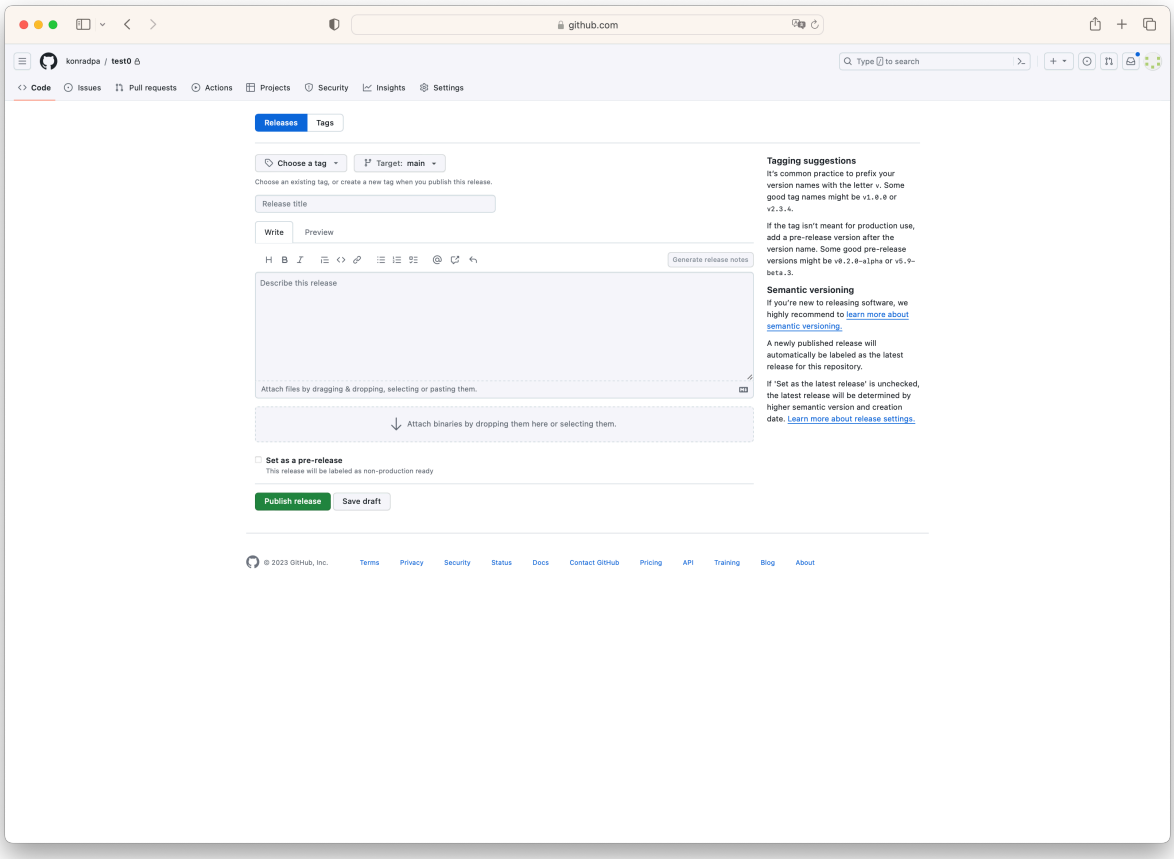



Figure 8.1: Screenshot of GitHub when creating a new release

and provide a persistent link to scholarly articles, research papers, books, datasets, and other types of digital content. The purpose of a DOI is to ensure that the content can be reliably located and accessed over time, even if the web address (URL) of the resource changes. Key features of DOIs include:

1. **Uniqueness:** Each DOI is unique to a particular resource, ensuring that no two resources have the same identifier.
2. **Persistence:** DOIs are designed to remain unchanged, providing a persistent link to the resource even if it is moved or the URL changes.
3. **Interoperability:** DOIs are widely used in scholarly publishing and other sectors, making them interoperable across different systems and platforms.
4. **Accessibility:** DOIs are often associated with metadata that provides information about the resource, such as author, title, publisher, publication date, and more.
5. **Citations:** DOIs are commonly used in academic and scientific citations to provide a standardized and reliable reference to a specific resource.

DOIs are typically assigned and managed by registration agencies, such as CrossRef for scholarly content or DataCite for research data. Organizations and publishers assign DOIs to their digital content to enhance discoverability, citation tracking, and long-term accessibility.

### 8.3.1 Creating a Zenodo account

To upload a Git repository to Zenodo, you first create a Zenodo account. On the Zenodo start page, you can click on "**Sign Up**" in the top right corner. Here you either can directly use your **GitHub account** to sign up, or sign up using **your email** (or other accounts) and link your GitHub account later on. To **link your GitHub account**, you can click on the arrow, next to your email, on the start page and select "GitHub".

### 8.3.2 Uploading a Git repository

If you are just trying out how to upload a Git repository, we recommend to use the Zenodo sandbox, to not create an unnecessary real DOI which is hard to delete. Your Zenodo account should work perfectly fine in the sandbox.

> ! **Note that you have to link your Zenodo account and sync your repository *before* you create a new release on GitHub!**

To upload an older release, you have to download your repository as a `.zip` file from GitHub and uploading it manually at https://sandbox.zenodo.org/uploads/new

To upload a new release, visit the Zenodo GitHub settings. Here you should see all your uploaded repositories, with the option to sync them by selecting the "on" button. If you now create a release, as discussed earlier, it will show up and be associated with a DOI.

For a pictured guide on how to upload a GitHub repository to Zenodo, you check out this guide from the Code Refinery.

## 8.4 Acknowledgements and further reading

We would like to express our gratitude to the following resources, which have been essential in shaping this chapter. We recommend these references for further reading:

| Authors | Title | Website | License | Source |
|---|---|---|---|---|
| Chacon and Straub [7] | Pro Git | | CC BY-NC | |
| coderefinery [8] | GitHub without the command line | | CC BY-NC 4.0 | |

## 8.5 Cheatsheet

| Command | Description |
|---|---|
| `git tag` | Lists all tags |
| `git tag v1.0` | Creates a tag on the basis of your current commit named `v1.0` |
| `git tag v1.1 <commit-hash>` | Creates a tag on the basis of a specific commit hash named `v1.1` |
| `git tag -a v1.0 -m "Release version 1.0"` | Creates an annotated tag on the basis of your current commit hash named `v1.0` with the tagging message `Release version 1.0` |
| `git push origin <tag-name>` | Pushes a specific tag to remote |
| `git push origin --tags` | Pushes all created tags to remote |
| `git fetch --tags` | Fetches all created tags from remote |
| `git pull --tags` | Pulls all created tags from remote |

# 9 Issues

UHH WS 23/24

Learning Objectives
- ☐ Understand the purpose of GitHub Issues
- ☐ Master the creation and management of Issues
- ☐ Collaborative problem solving with Issues
- ☐ Practicing with a practical exercise

## 9.1 Overview

GitHub is famous for helping people work together on code. But to work well together, you need to talk and organize tasks. GitHub makes this easy with a feature called GitHub Issues. This chapter shows you **how to use GitHub Issues for organizing and following a task**.

GitHub issues are a powerful and often used feature of the platform. Think of the issues for a project as its bug tracker. Even for projects that are not pure software development, we can co-opt this machinery to organize our to-do list more generally.

The basic unit is an issue and you can interact with one in two ways. First, issues are integrated into the project's web interface on GitHub, with a set of options for linking to project files and incremental changes. Second, issues and their associated comment threads appear in your email, just like regular messages (this can, of course, be configured). The result is that all correspondence about a project comes through your normal channels, but is also tracked inside the project itself, with good navigability and search capabilities. For software, issues are used to track bugs and feature requests. In a data analysis project, you might open an issue to flesh out a specific sub-analysis or to develop a complicated figure. In a course, we could use them to manage homework submission, marking, and peer review.

Issues can be assigned to specific people and they can be labeled, for example, `bug`, `simulation-study`, or `final-exam`. Coupled with the ability to cross-link issues and the project files or file changes, you have the power to document why things have happened in the past and to organize what needs to happen in the future.

## 9.2 Benefits of issues

### Organized research tasks

Issues help scientists keep track of their research tasks, experiments, and observations in an organized and structured manner.

**Easier collaboration**

Issues facilitate seamless collaboration among scientists, allowing them to work together on experiments, data analysis, and research projects regardless of geographical locations.

**Clear Communication**

Scientists can use issue descriptions and comments to communicate their findings, hypotheses, and insights, fostering a transparent and clear dialogue within their research teams.

**Structured Problem Solving**

Issues enable scientists to break down complex research problems into smaller, manageable tasks, making it easier to solve challenges step by step.

**Documentation and Records**

By creating issues, scientists create a comprehensive record of their research progress, methodologies, and results, providing valuable documentation for future reference and publication.

**Task prioritization**

Researchers can assign labels and milestones to issues, helping them prioritize tasks, experiments, and analyses based on urgency and importance.

**Traceable Changes**

Scientists can track changes made to issues, allowing them to see the evolution of ideas, experiments, and conclusions over time.

**Error Tracking and Resolution**

Issues are ideal for identifying, documenting, and resolving errors, ensuring accuracy and reliability in scientific experiments and data analysis.

## 9.3 Features - Overview

- You can organize issues by labels, for example `bug`.
- You can assign responsibility for an issue to a team member.
- You can add the issue to a project milestone.
- Issues can be combined in Issue boards
- Issues can be sorted (by due date, label priority, etc.)
- Issues can be transferred between repositories
- Issues can be crosslinked e.g., in commit messages: `git commit -m "add missing data, close #37"`
- Issues can send automated email notifications
- Issues can be exported and archived

(more details here)

## 9.4 Step-by-step

### 9.4.1 Prerequisites

To make an issue, **you need a repository**. You have the option to use a repository you already have permission to write in, or you can create a new repository. The repository must have issues enabled. If you want more guidance on making a repository, refer to the chapter on first steps with Git. If you need help with turning on issues in case they're currently turned off in your repository, take a look at the section titled "disabling issues".

### 9.4.2 Opening a blank issue

First, create a new issue. There are multiple ways to create an issue. You can choose the most convenient method for your workflow. This example will create an issue from a repository on GitHub. For more information about other ways to create an issue, see the section "Creating an issue" in the GitHub documentation.

1. On GitHub, navigate to the main page of the repository. For example, the main page of the GitHub repository of this guide can be found here.
2. Under the repository name, click **Issues** (see Figure 9.1).
3. On the following page, click the green **New issue** button.

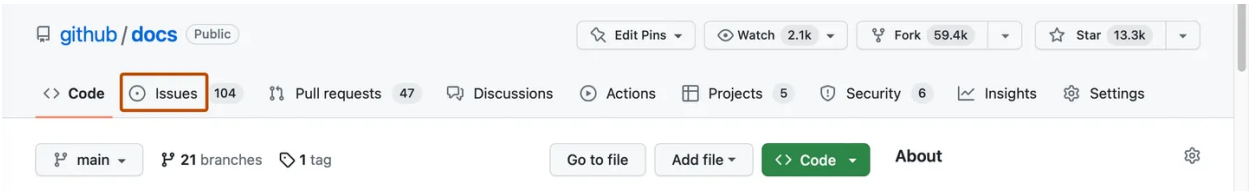



Figure 9.1: "Repo Tabs Issues" taken from the GitHub Docs (docs.github.com). The GitHub product documentation in the assets, content, and data folders are licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence (direct link).

> **Try it!**
>
> Open a new Issue in the repository of this online guide (requires a GitHub account).

If the repository uses issue templates (details below), click the green **Get started** button next to the type of issue you'd like to open (see Figure 9.2). If the kind of issue you want to create isn't listed among the choices, simply click on "Open a blank issue".

### 9.4.3 Filling in issues

1. Choose a descriptive **title** for your issue. The title should convey what the issue is about.

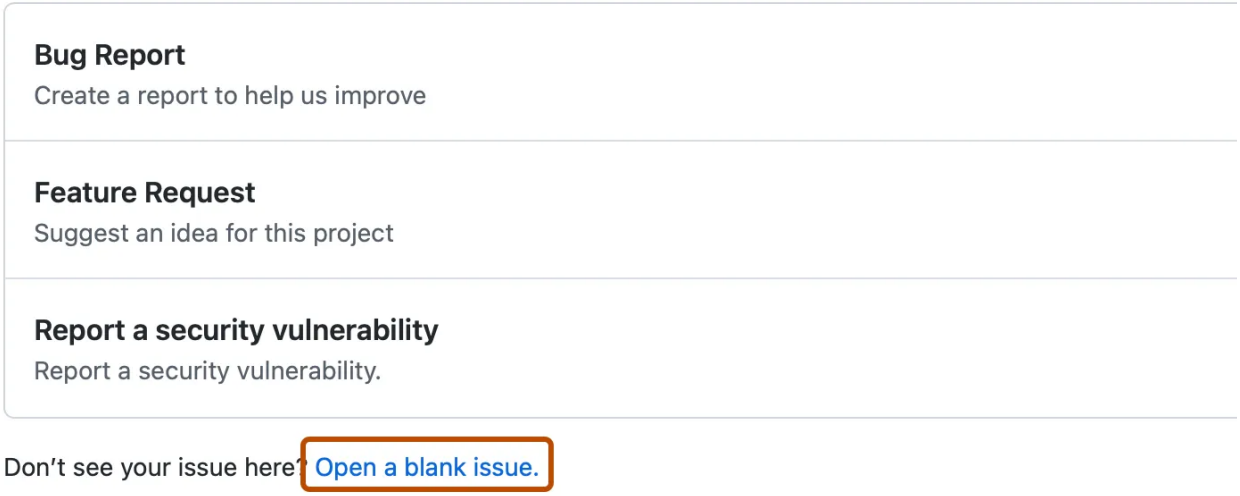



Figure 9.2: "Blank Issue Link" taken from the GitHub Docs (docs.github.com). The GitHub product documentation in the assets, content, and data folders are licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence (direct link).

2. Include a **description** that explains why you're creating this issue and any information that might help to solve it. For instance, if it's about a bug, explain how to reproduce the bug, what you expected to happen, and what actually happened. You have the option to use Markdown to add special formatting, links, emojis, and other things. For more information, see the section "Writing on GitHub" in the GitHub documentation.
3. You can also attach **files** or **images** by dropping, selecting or pasting them into the issue.
4. When you're done, click the green **Submit new issue** button. You can still edit any of the fields after creating the issue.

On the right-hand side, you'll see options to assign someone to the issue, add labels, add the issue to projects, or add the issue to milestones. We'll explore these features in more detail below. You can also set all of these options after you submitted the issue.

### 9.4.4 Submitting your issue

Click **Submit new issue** to create your issue. You can edit any of the above fields after creating the issue. Your issue has a unique URL that you can share with team members, or reference in other issues or pull requests.

## 9.5 Features of Issues

### 9.5.1 Editing issues

You can always edit the title and description of your issue after you submitted the issue.

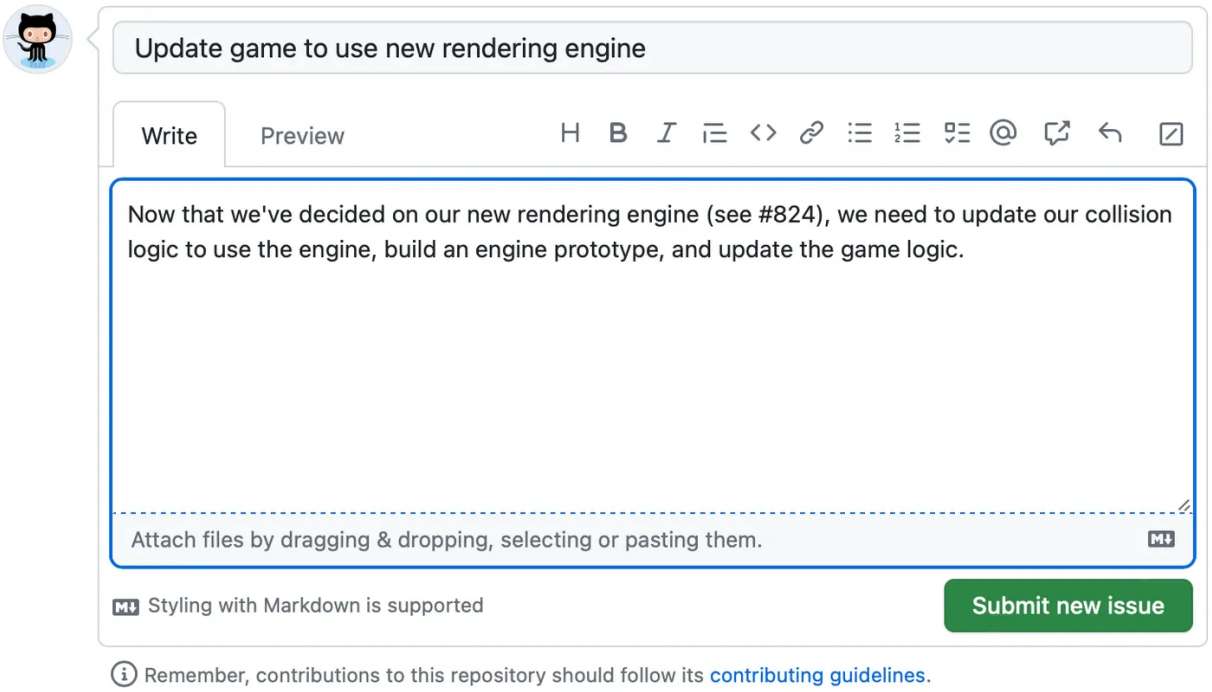



Figure 9.3: "Issue Title Body" taken from the GitHub Docs (docs.github.com). The GitHub product documentation in the assets, content, and data folders are licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence (direct link).

On the right side of the title, you'll notice an **Edit** button. This comes in handy if you ever need to change the title of the issue as the discussion evolves over time or you want to make the title more descriptive. Even if you edit the issue, the issue number will always remain the same.

What if you decide to modify the main text of your issue or a comment after you've submitted it? No problem at all. Once your issue is submitted, a blue bar will appear at the top of your comment. This bar displays your username and the comment's date. Toward the right side of this blue bar, you'll notice three dots. If you click on those dots, you'll find the option to edit your comment.

### 9.5.2 Communicating in issues

#### 9.5.2.1 Commenting issues

The major benefit of issues is that **they serve as platforms for conversations with yourself and others**. Once an issue has been submitted, you can continue the conversation by adding more comments within that issue. For example, you might ask a question, and someone else could respond with a solution or idea. They might even include links to related issues or external resources that could be helpful.

#### 9.5.2.2 Mentioning people

You can also **mention people in issue comments** using the `@` symbol. Anyone who is part of the repository will automatically receive email notifications about new comments. Tagging specific users with the `@` symbol will also send them an email, making it a useful way to involve individuals who might not already be closely following the entire repository. In a public repository, you can tag any GitHub user, while in a private repository, they need to have appropriate permissions.

To **connect related issues in the same repository**, you can type # followed by part of the issue title and then clicking the issue that you want to link.

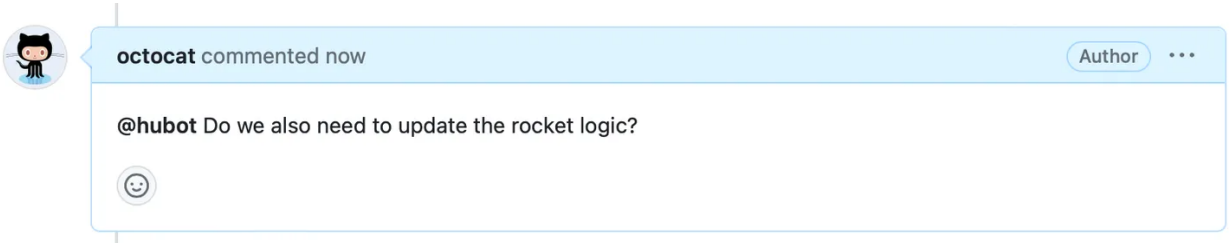



Figure 9.4: "Issue Comment" taken from the GitHub Docs (docs.github.com). The GitHub product documentation in the assets, content, and data folders are licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence (direct link.

#### 9.5.2.3 Timeliness of issues

Whenever a new comment is added to the issue thread, a new date will be displayed in the blue bar at the top of the issue. This provides a convenient way to assess the timeliness of ongoing discussions.

#### 9.5.2.4 Referencing comments

Another useful feature is the ability to click on the date of an issue's comment, causing the URL in your browser to update. This updated URL lets you **link directly to a particular comment in the issue thread**. This is very handy when you want to share a specific comment with someone, saving them the trouble of scrolling. Alternatively you can also you can also click on the three dots on the right of the blue comment header and select the option to copy the link for that specific comment.

#### 9.5.2.5 Using Markdown syntax

GitHub allows you to use Markdown syntax in your issue comments, which provides an easy way of formatting text. For example, you can create headers using the # symbol (`# Header 1`, `## Header 2`, and so on). For a complete overview of Markdown syntax, you cn check out this GitHub documentation.

### 9.5.3 Adding a task list

One way of using Markdown syntax in GitHub issues, is **creating a task list**. It can be helpful to divide big problems into smaller tasks or group related issues together. A task list is a set of tasks that each appear on a separate line with a clickable checkbox. You can select or deselect the checkboxes to mark the tasks as complete or incomplete. You can make a task list by putting `- [ ]` in front of list items. You can refer to other issues using their number or URL. You can also use plain text to keep track of tasks without issues and turn them into issues later if needed. For more information, see the section "About task lists" in the GitHub documentation.

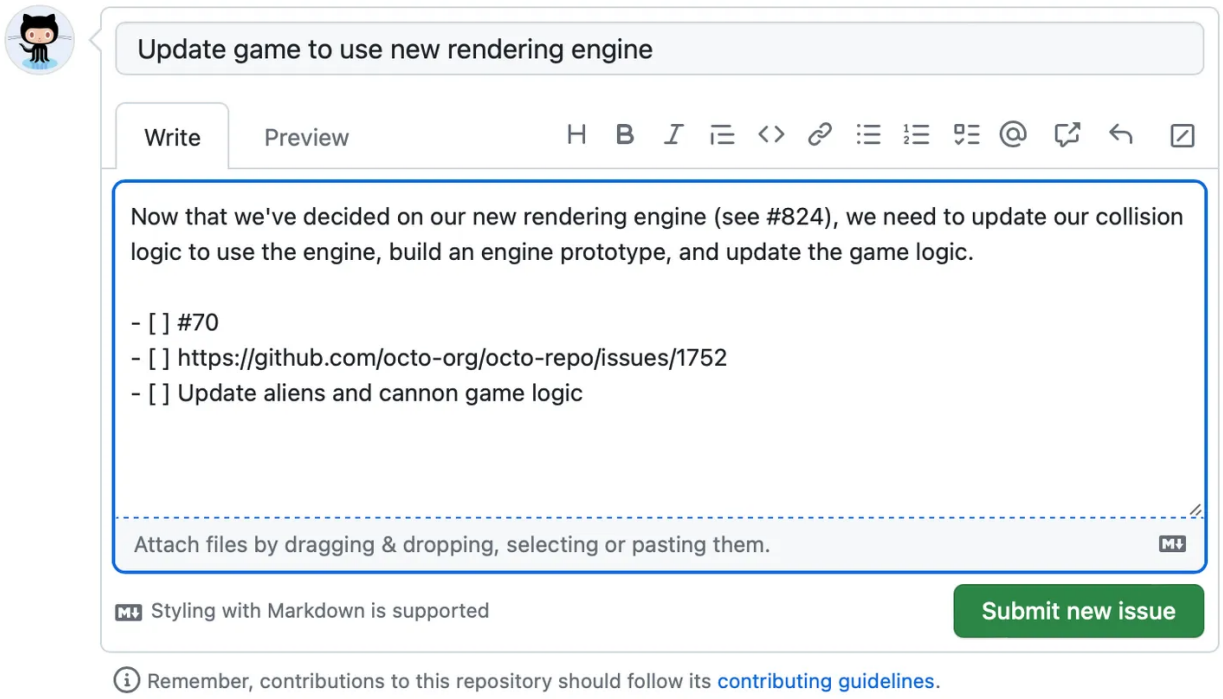



Figure 9.5: "Issue Task List Raw" taken from the GitHub Docs (docs.github.com). The GitHub product documentation in the assets, content, and data folders are licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence (direct link).

### 9.5.4 Adding labels

Add a label to **categorize your issue**. You can apply labels to an issue on the right side of the issue thread or when creating a new issue (see Figure 9.6). For example, you could apply a `bug` label and a `good first issue` label to indicate that an issue is a bug that a first-time contributor could tackle. Users can filter issues by label to find all issues that have a specific label. For example, you can find a list of issues for this repo here. For information, see the section "Managing labels" in the GitHub documentation.

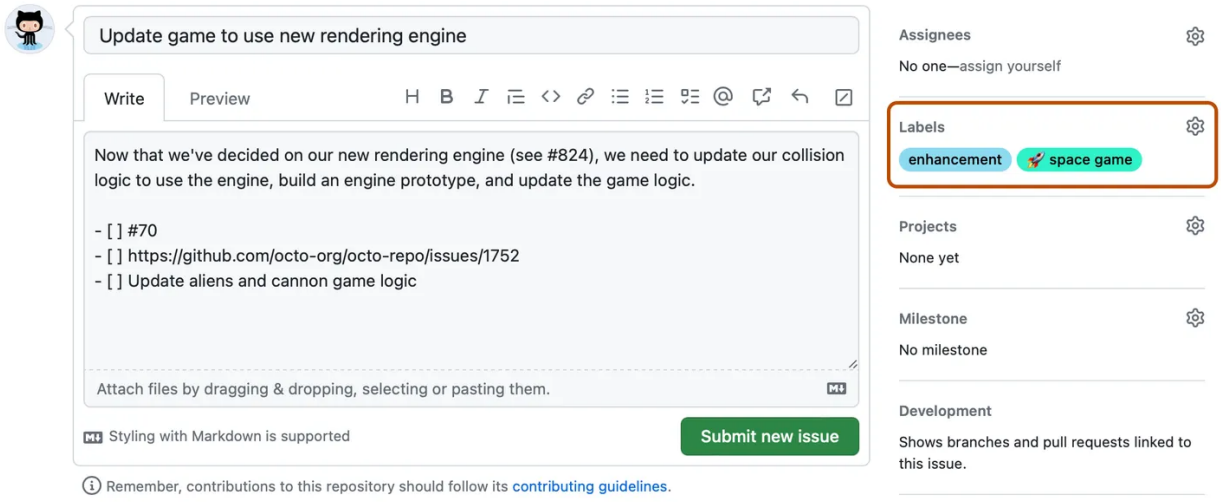



Figure 9.6: "Issue With Label" taken from the GitHub Docs (docs.github.com). The GitHub product documentation in the assets, content, and data folders are licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence (direct link).

#### 9.5.4.1 Creating a label

Anyone with write access to a repository can **create a new label** or **customize existing labels**.

1. On GitHub, navigate to the main page of the repository.
2. Under the repository name, click **Issues** or **Pull requests**.
3. Above the list of issues or pull requests, click **Labels**.
4. To the right of the search field, click **New label**.
5. Under "Label name", type a name for your label.
6. Under "Description", type a description to help others understand and use your label.
7. Optionally, to customize the color of your label, edit the hexadecimal number, or, for another random selection, click the rotating arrows.
8. To save the new label, click **Create label**.

### 9.5.5 Adding milestones

You can add a milestone to **track the issue as part of a higher-order project target**. A milestone will show the progress of the issues as the target date approaches. For more information, see the section "About milestones" in the GitHub documentation.

### 9.5.6 Assigning the issue

To clearly **communicate responsibility for an issue**, you can assign the issue to a team member of your project or organization.

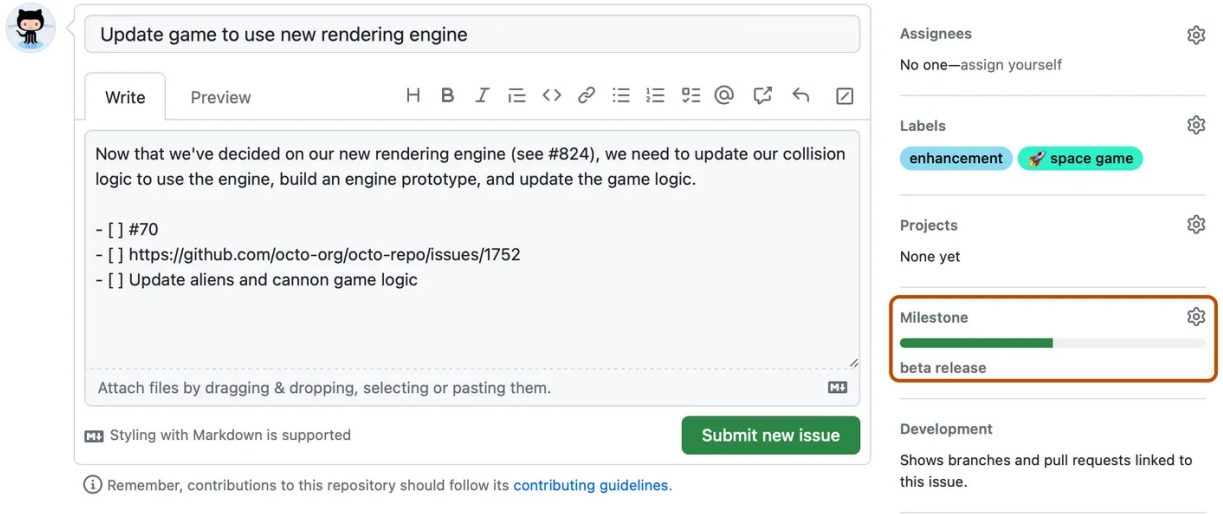



Figure 9.7: “Issue Milestone” taken from the GitHub Docs (docs.github.com). The GitHub product documentation in the assets, content, and data folders are licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence (direct link).

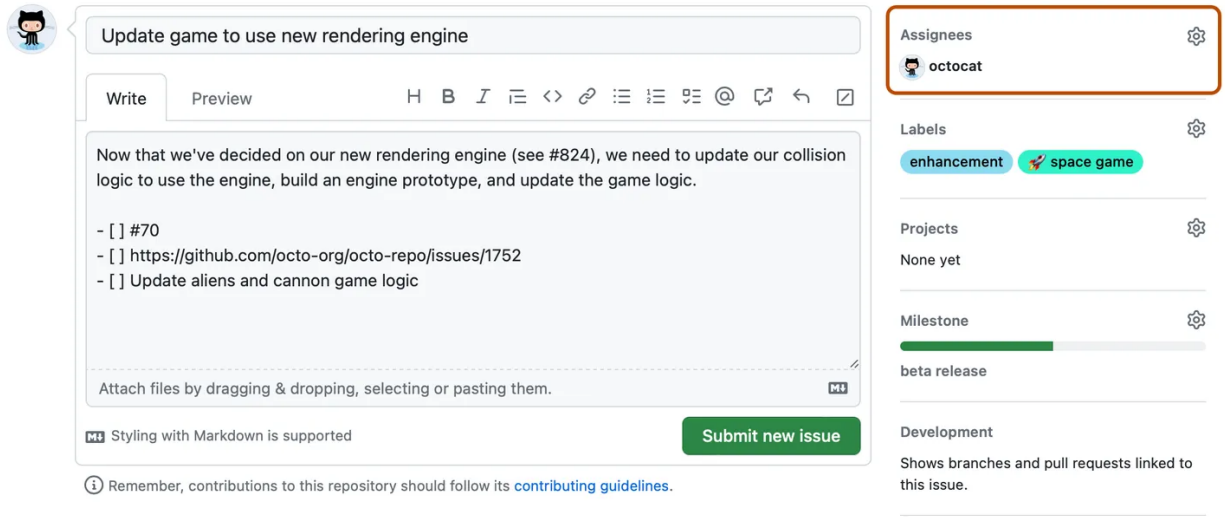



Figure 9.8: “Issue Assignees” taken from the GitHub Docs (docs.github.com). The GitHub product documentation in the assets, content, and data folders are licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence (direct link.

### 9.5.7 Adding projects

You can add the issue to an existing project and populate metadata for the project.

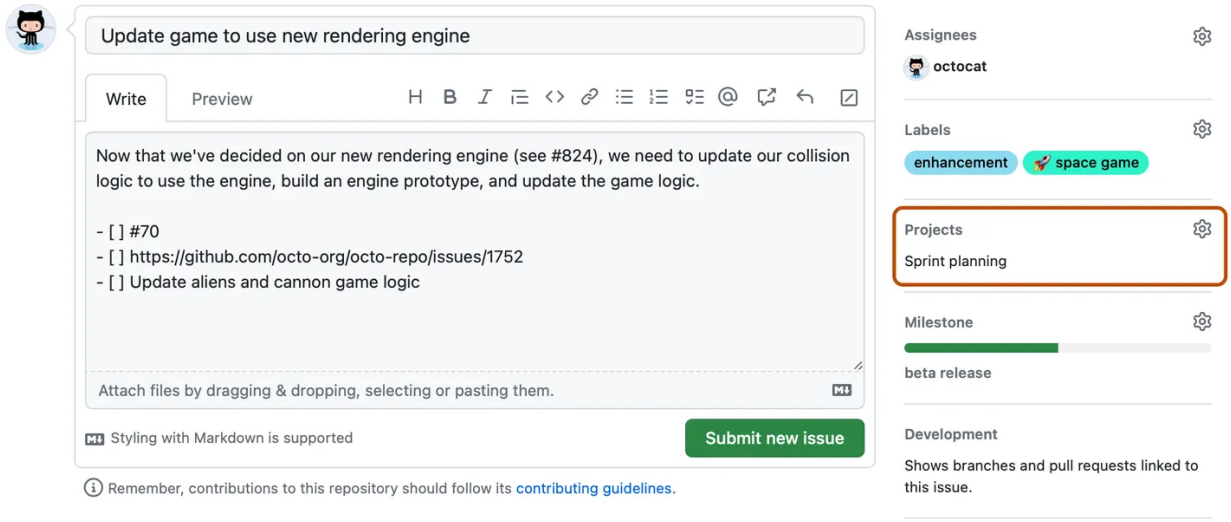



Figure 9.9: "Issue Project" taken from the GitHub Docs (docs.github.com). The GitHub product documentation in the assets, content, and data folders are licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) licence (direct link.

Learn more about projects

### 9.5.8 Issue templates

GitHub issue templates are **predefined formats for creating new issues** in a repository. They help with the process of opening new issues by providing a structure for the information that contributors should include when reporting a bug, requesting a feature, or any other type of issue. You can create multiple templates for different types of issues, such as bug reports, feature requests, or general inquiries Each template is a Markdown file (`.md`) with predefined sections, such as description, steps to reproduce, expected behavior, actual behavior, etc.

When a user opens a new issue, GitHub automatically displays the template content in the issue description field. Users can then fill in the required information based on the template.

To create a new template follow these steps:

1. Navigate to the repository on GitHub.
2. Click on the **Settings** tab.
3. Under the General tab, in the **Features** section, find the Button labeled: **Set up templates**

Here you will find predefined templates by GitHub, but also the option to create a new issue template from scratch.

### 9.5.9 Linking to an issue

You can link to an issue directly from the command line! You can link a pull request, branch or commit to an issue to show that a fix is in progress and to automatically close the issue when the pull request, branch or commit is merged. For example:

```
git commit -m "Add a new file. close #7"
```

This closes issue number 7 of your repository. (if committed to the default branch)

#### 9.5.9.1 Linking to an issue using a keyword

You can link to an issue by using a supported keyword in the pull request's description or in a commit message. The pull request or commit must be on the default branch. Possible keywords are:

- close
- closes
- closed
- fix
- fixes
- fixed
- resolve
- resolves
- resolved

For further information, see this chapter of the GitHub docs.

### 9.5.10 Disabling issues

If you want to disable issues for your repository, you can easily do this in your repository's settings.

1. Navigate to the repository on GitHub.
2. Click on the **Settings** tab.
3. Under the General tab, in the **Features** section, find the **checkbox** related to issues.
4. **Uncheck** or disable the issues feature.

## 9.6 Strategies for Issues

Most repositories on GitHub have Issues. But should you use Issues in every project? It's helpful to think about why you're using them.

### 9.6.1 Issues for bug reporting and feature requests

If you're using Issues for typical things like reporting and fixing problems in code or adding new features, it's a good idea to keep the project open to the public. You can have all the discussions related to that project inside it.

### 9.6.2 Issues for internal conversations

If you're using Issues for internal conversations (for example, discussions within a research group), you might want to keep the repository private at first. But here's something to consider: Do you plan to make the project public when you're ready to share your research with the world? Changing a project from private to public (or the other way around, which you can do in the project's settings) not only makes the code and files public but also all the discussions in the Issues. That's okay, but it adds some things to think about in terms of what's being talked about in those discussions.

## 9.7 Further Reading

- GitHub Docs: Tracking your work with issues
- GitHub for project management - openscapes

## 9.8 Acknowledgements

| Authors | Title | Website | License | Source |
|---|---|---|---|---|
| Lowndes and Robinson [11] | Openscapes Champions Lesson Series |  | CC BY 4.0 |  |

# 10 Graphical User Interfaces

In this chapter, we will give an introduction to Graphical User Interfaces (GUIs) for Git. We will showcase two different, commonly used GUIs, including GitHub Desktop and GitKraken as well as the Git integration in RStudio. Finally, we will discuss specific use cases where a graphical user interface makes Git operations easier and less error-prone compared to the command-line.

Learning Objectives
- Understanding the benefits of Git GUIs
- Exploring different GUIs
- Exploring branch management in a GUI
- Practicing with a practical exercise

Exercises
Install GitKraken or GitHub Desktop Login to the client using your GitHub Account and view your recipe repository Add two recipes in the same file, stage and commit only one of them, using a GUI Open your recipe repository in RStudio, edit a file and commit your changes

## 10.1 Introduction to Git GUIs

Graphical User Interfaces (GUIs) offer a more user-friendly way to work with Git compared to the command-line, using visuals instead of text commands. They show your project's history, branches, and changes in a more colorful and easy-to-understand manner.

They make Git seem less scary and more accessible to everyone, helping you manage your code, without the hassle of remembering complex commands. While Git GUIs offer a user-friendly approach to version control, they come with a few limitations. One notable drawback is their reduced flexibility compared to the command-line interface. GUIs are designed to simplify common tasks, but more advanced or customized operations may be challenging to perform within a graphical environment. Despite their user-friendly design, understanding the underlying Git concepts and the specific workflow of a GUI is still very important.

As this book primarily emphasizes teaching Git through the command line interface, it will only feature a short introduction, not extensive tutorials for GUI tools.

## 10.2 Popular Git GUIs

There is a wide variety of Git GUIs available, each of them offers its own set of features and benefits, catering to different preferences and workflows. In this section, we will focus on GitHub Desktop and GitKraken, although many more options exist. For a more comprehensive overview, you can explore the official Git website.

### 10.2.1 GitHub Desktop

#### 10.2.1.1 Download

GitHub Desktop is a desktop application developed by GitHub. It is free to download for macOS and Windows on the GitHub website. There is currently no official version available for Linux.

#### 10.2.1.2 Setup

After you download and install the program, you will be required to log in using your GitHub account. Once logged in, you can select a repository to work on, either from GitHub or a local repository (as shown in Figure 10.1).

Once you've picked a repository to work on, GitHub Desktop shows you the history of that repository in a visual way. You'll see branches, commits, and tags displayed on a timeline in chronological order. This setup makes it easy to dive into version control tasks like handling branches, commits, changes, and pull requests.

For a more extensive GitHub Desktop tutorial, you can check out the GitHub documentation.

In general, GitHub Desktop's primary advantage lies in its integration with github.com. The interface is simple and user-friendly. A significant drawback of relying on GitHub Desktop is its strong tie to the GitHub platform. This dependency can be limiting if you're looking to work with repositories on alternative Git hosting services.

### 10.2.2 GitKraken

#### 10.2.2.1 Download

GitKraken is a third-party program and another very popular git GUI client for Windows, macOS and Linux. You can download it from the official website. Normally, GitKraken only offers a free seven-day trial period. However, if you are a student, you can get it for free, through the GitHub Student Developer Pack.

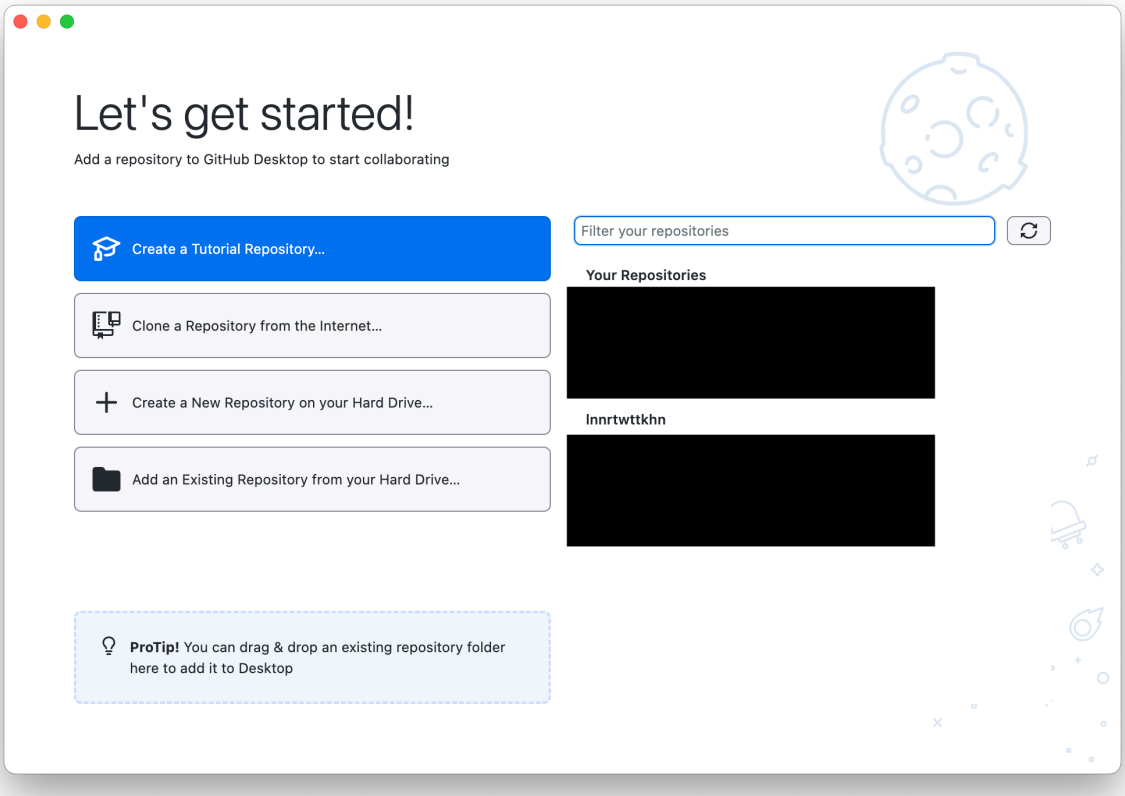



Figure 10.1: Screenshot of GitHub Desktop. Showing the start screen.

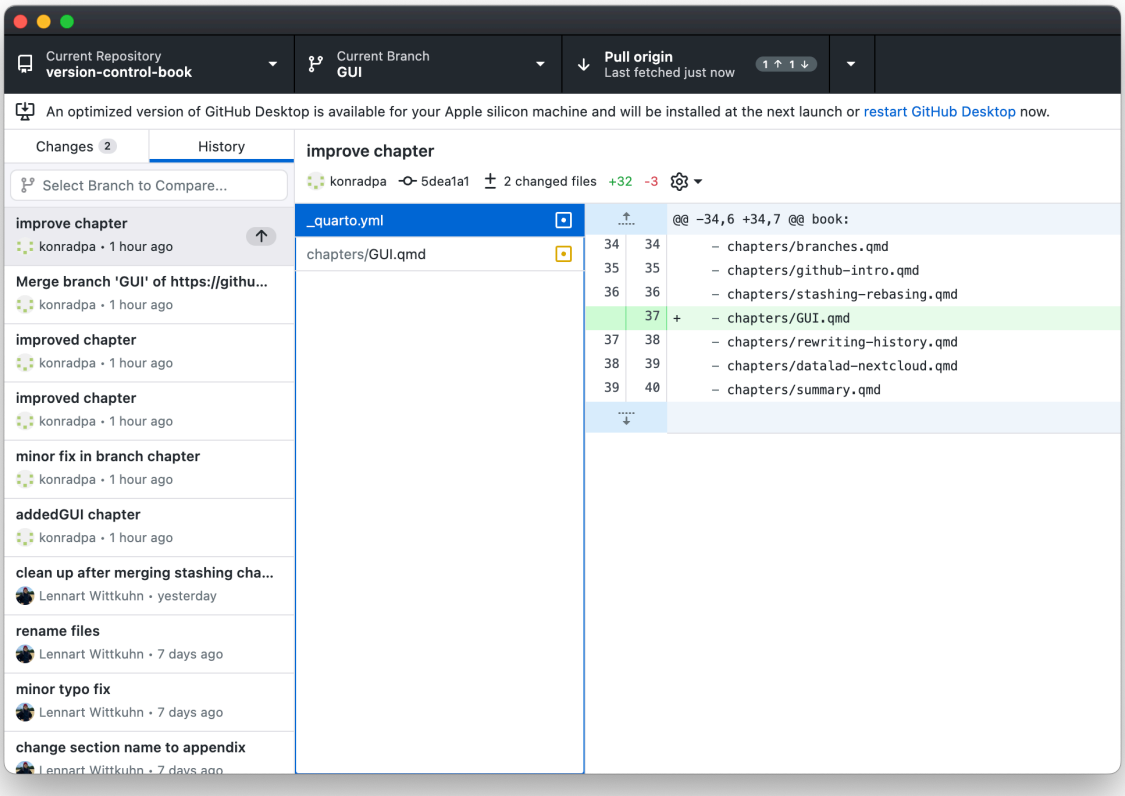



Figure 10.2: Screenshot of GitHub Desktop. Showing an example repository.

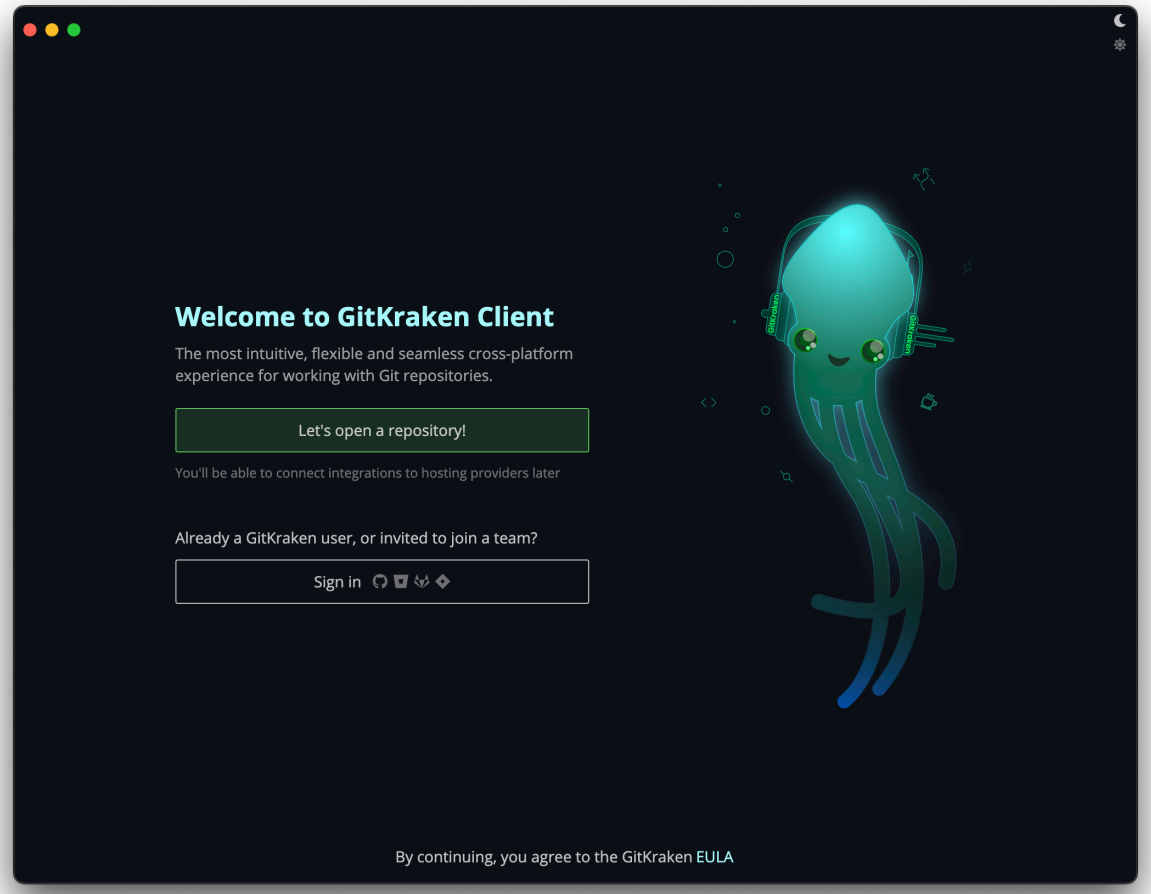



Figure 10.3: Screenshot of GitKraken. Showing the start screen.

#### 10.2.2.2 Setup

After downloading, you either begin to track a local repository or log in to a hosting website like GitHub (but also to other ones) and work with your repositories from there.

After choosing a repository to track, you will see a visual representation of the commit history, branches, and commits.

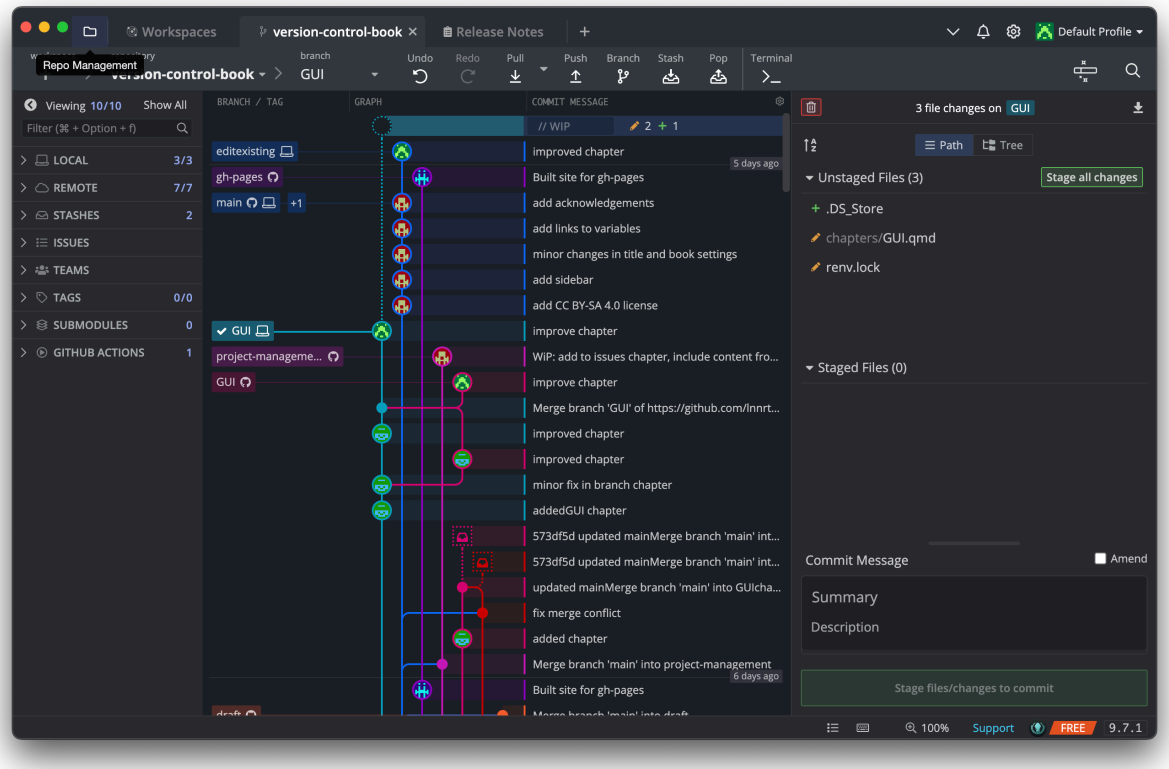



Figure 10.4: Screenshot of GitKraken. Showing an example repository.

Just like with GitHub Desktop, your are able to manage your branches, push to or pull from remote branches and commit changes through GitKraken. For an extensive tutorial, you can check out the GitKraken Client Documentation.

## 10.3 Use cases for Git GUIs

You can perform a wide range of basic and advanced Git operations with GUIs. Here, we present a few Git use cases where the use of a GUI can be particularly helpful as compared to doing the same thing from the command-line.

### 10.3.1 Partial commits

One useful feature of Git GUIs is the ability to easily commit only specific parts or “hunks” of code from within a file. This is particularly handy when you’ve made changes to different parts of a file and want to commit them separately to group related changes together and keep a clean commit history. With a Git GUI, you can easily select the lines or blocks of code you want to include in a commit, stage them, and then commit just those changes.

For example, to do this in GitKraken, change a file, then click on your last commit in the GitKraken GUI (see Figure 10.5). You will see a list of unstaged files with changes. After you click on one, you will see your additions highlighted green and your deletions highlighted red. If you hover over the changed lines you will see the option to “Stage this Line”. To stage multiple lines at the same time, highlight the ones you want to stage, do a right-click and select “Stage selected lines”. You can also stage the whole file or “hunk” and unstage specific changes.

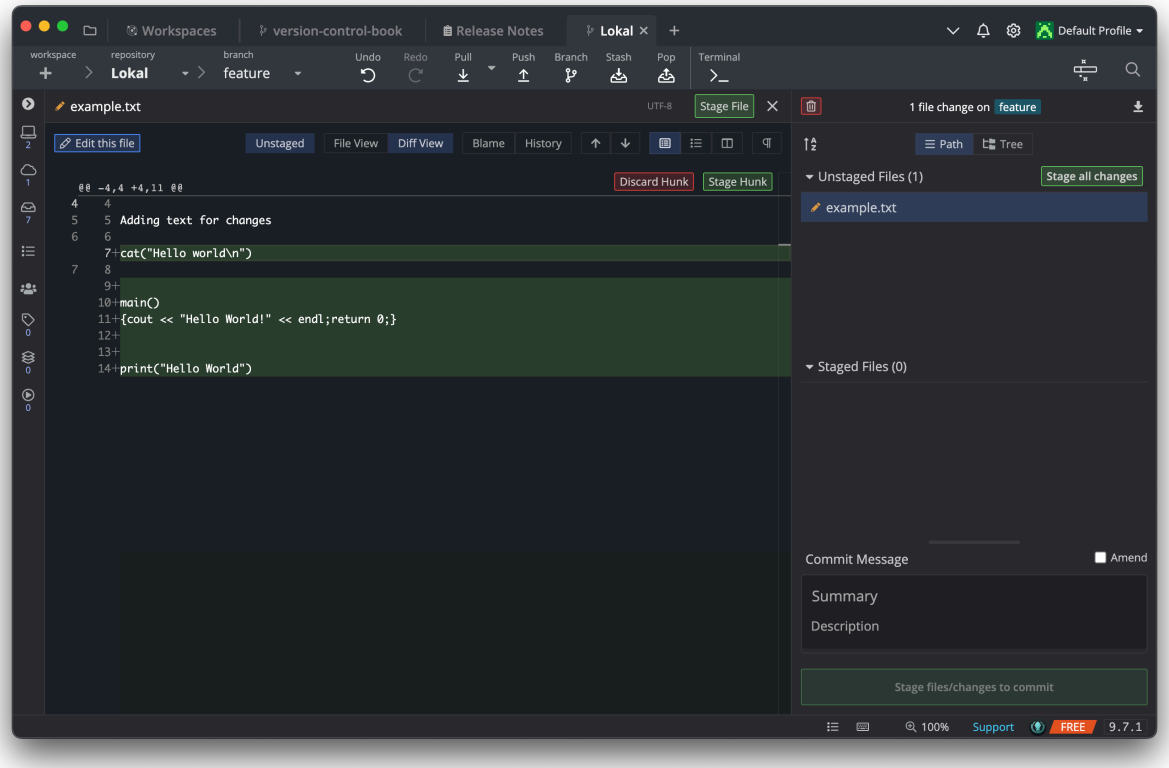



Figure 10.5: Screenshot of Rstudio. Showing staging an example file.

> **What is a hunk?**
>
> In Git, a “hunk” refers to a distinct block of code changes within a file. It represents a cohesive set of added, modified, or deleted lines in a specific location. Git automatically divides changes into hunks to facilitate easier review, selective staging, and conflict resolution during version control operations.

You can also create a partial commit in GitHub Desktop. For details, see the GitHub Desktop

documentation.

### 10.3.2 Merge conflict resolution

Git GUIs make resolving merge conflicts more intuitive. When conflicts arise due to incompatible changes made in different branches, a GUI helps you identify conflicting lines side by side. You can then select which changes to keep, discard, or modify. The graphical representation simplifies the process.

To merge two branches in GitKraken, you can either drag and drop one branch onto another branch or use the right-click menu of the branch you want to merge in. If a merge conflict occurs you will see a list of conflicted files. When you click on a file, it will show you a side-by-side comparison of the conflicting changes, allowing you to choose which lines to keep. After resolving all conflicts, you can continue the merge process by completing the merge commit.

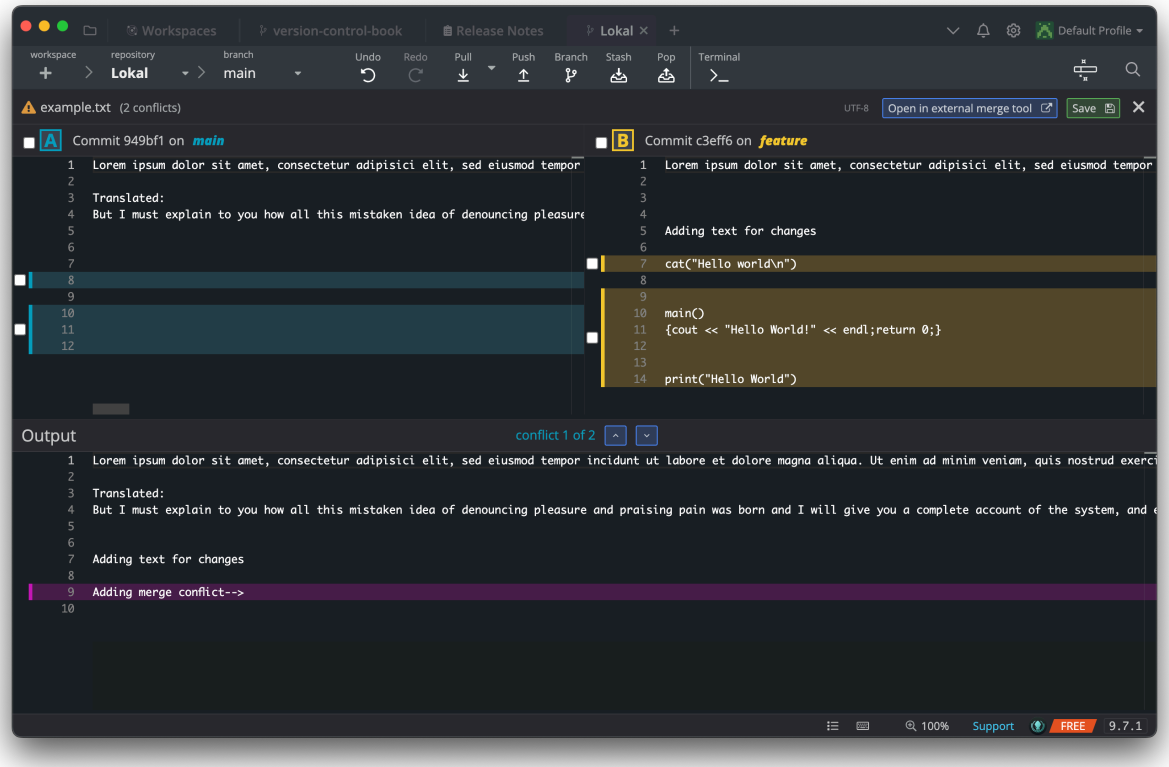



Figure 10.6: Screenshot of Gitkraken. Showing an example merge conflict.

## 10.4 Git Integration in RStudio

Even though it is not a complete GUI, RStudio Desktop, the most popular development environment for R programming, offers Git integration to simplify version control within your projects.

When opening a new RStudio project, you can choose to “Checkout a project from a version control project”. After choosing this, you have the option to clone a Git repository, for example from Github. RStudio will then download the files and you can start to edit them in RStudio.

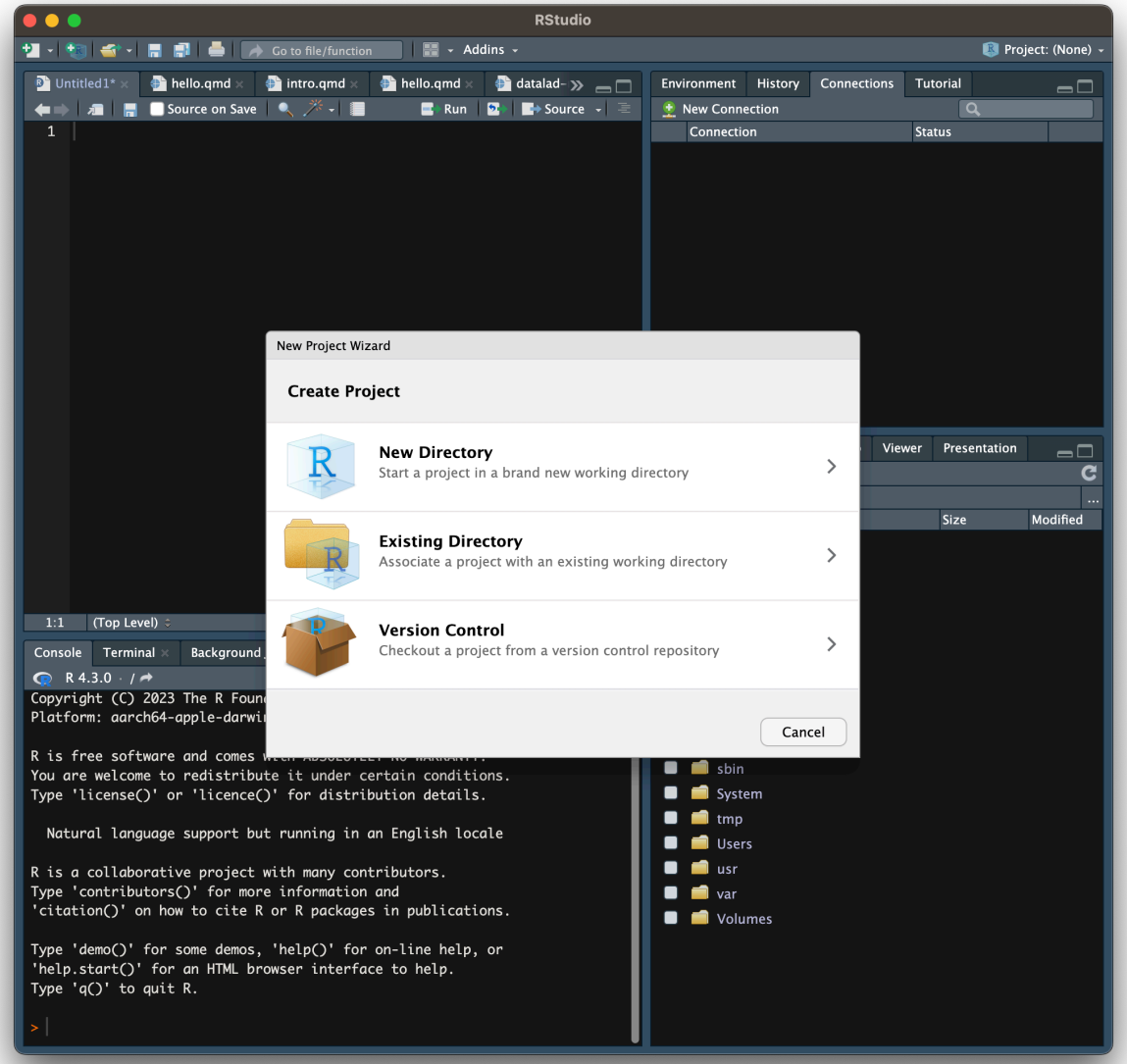



Figure 10.7: Screenshot of Rstudio. Showing the options for loading a repository.

After cloning a remote repository or opening a local Git repository folder, you can do things like committing, pushing or switching branches using buttons in Rstudio, instead of using the command line. The Git tab within the upper right panel enables you to stage, commit, push and pull changes. The branches tab in the Git pane displays a list of available branches, making it straightforward to switch between them.

Clicking “Commit” will open up a window, where you see your changes marked with colors. You also easily write commit message and have the option to amend your last command. It is also possible to stage specific lines.

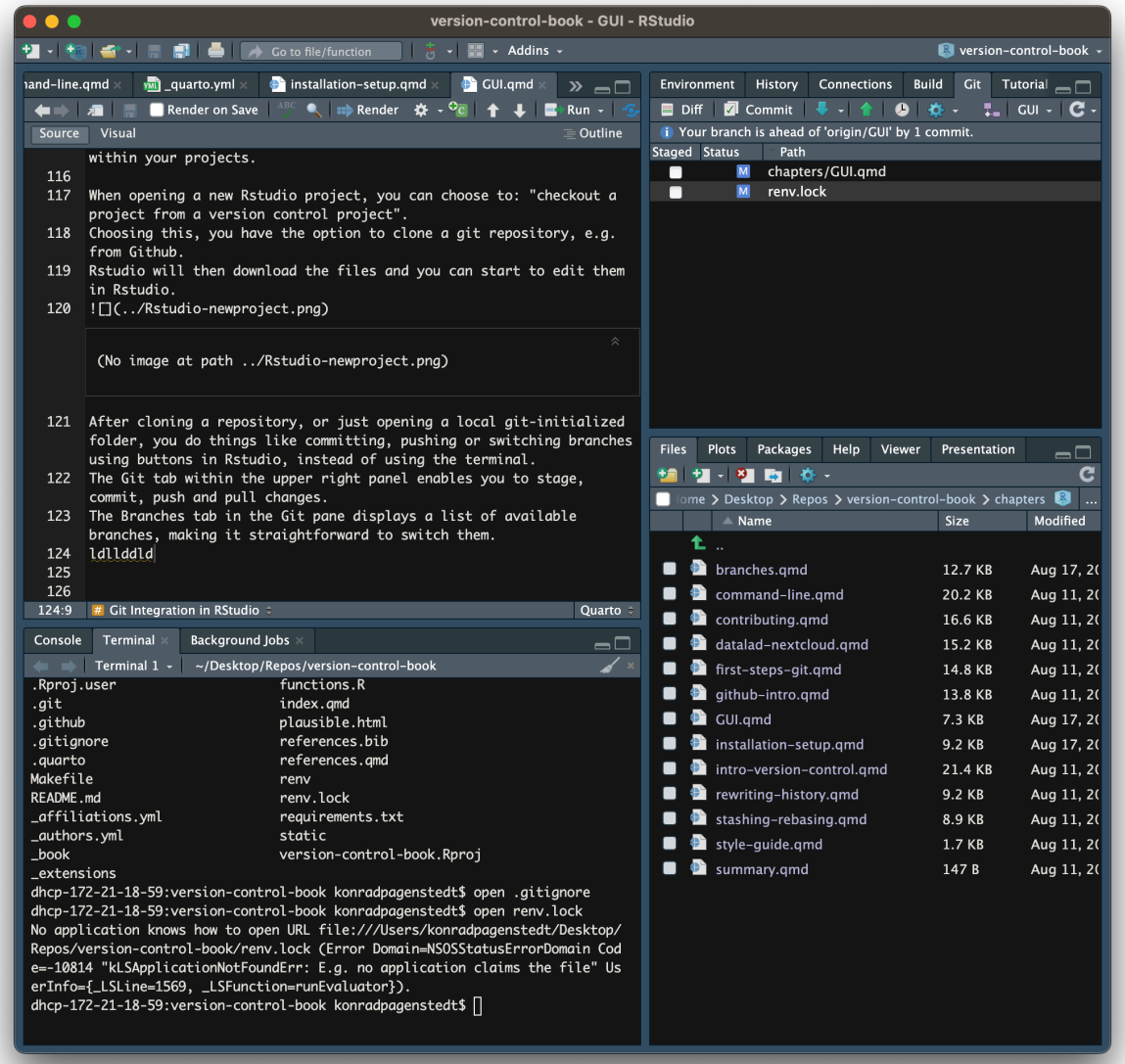



Figure 10.8: Screenshot of Rstudio. Showing an example repository.

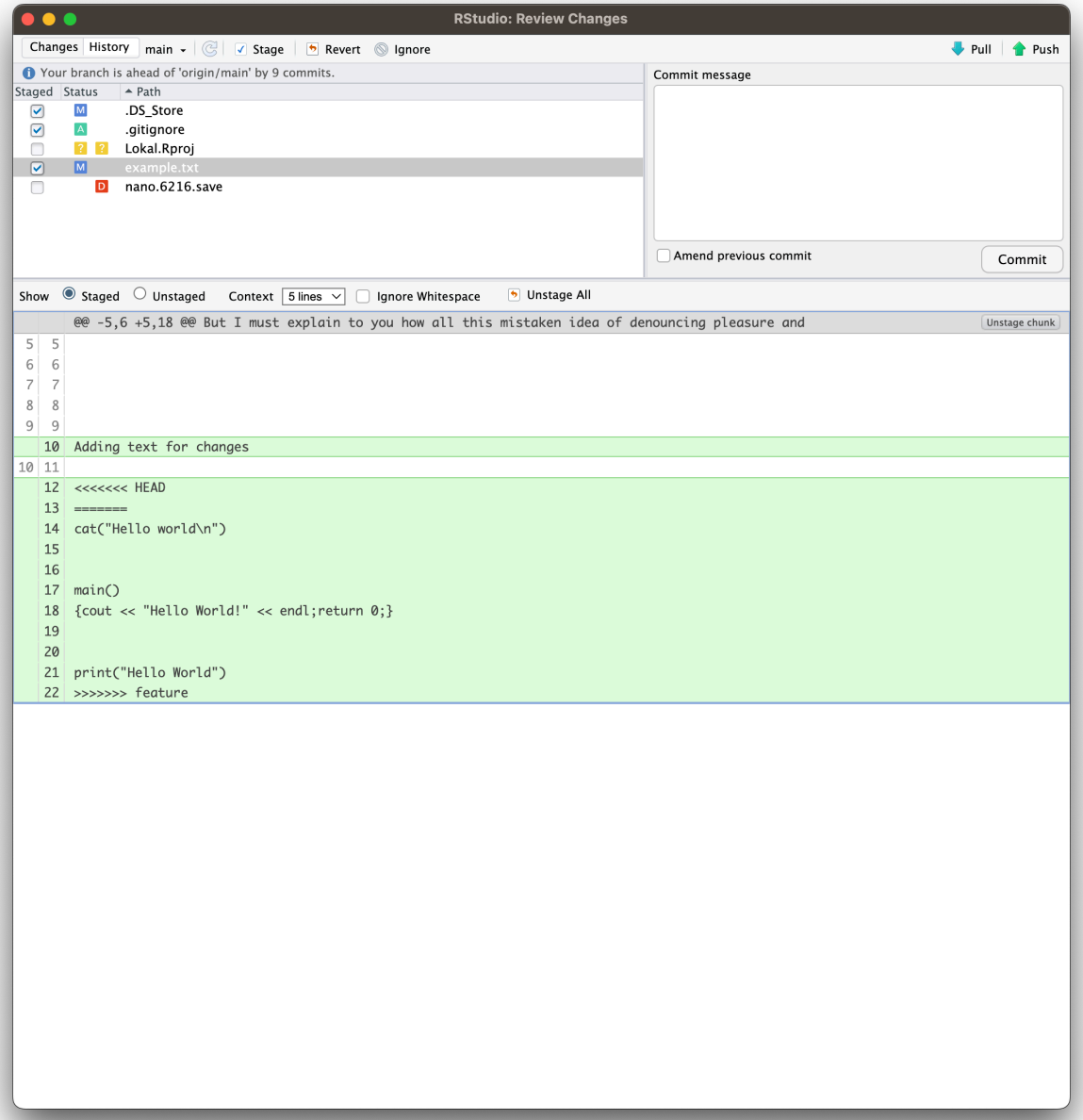



Figure 10.9: Screenshot of Rstudio. Showing an example commit.

For a more comprehensive tutorial on using Git in Rstudio you can check out the "Happy Git and GitHub for the useR" by Jenny Bryan.

You should keep in mind that while the RStudio GUI simplifies many Git tasks, it lacks the flexibility and customization of using the command line. But considering you can also open a command line tab in RStudio, the GUI might be a help in your workflow, if you work on a project and track it using Git.

## 10.5 Acknowledgements & further reading

| Authors | Title | Website | License | Source |
|---|---|---|---|---|
| Bryan [5] | Happy Git and GitHub for the useR | | CC BY-NC 4.0 | |
| GitHub [9] | GitHub Docs | | CC BY-NC 4.0 | |

# 11 Stashing and Co.

In this chapter, we will look at some more advanced essential commands, including stashing, reverting and rebasing

Learning Objectives
- ☐ Stashing and retrieving changes
- ☐ Undoing changes and removing files
- ☐ Rebasing and cherry-picking

Exercises
- ☐ Stashing
  - Create a new branch called `feature/stash-exercise`.
  - Make changes to two different files in your project directory.
  - Stash your changes without adding a message.
  - Use `git status` to verify that your working directory is clean.
  - Apply the stash to your working directory and verify that your changes are restored.
- ☐ Reverting
  - Create a new branch called `feature/revert-exercise`.
  - Make changes and commit these to your file.
  - Use `git revert` to revert the most recent commit, specifying its hash.
- ☐ Rebasing
  - Create a new branch called `feature/rebase-exercise`.
  - Make three commits with minor changes on the `feature/rebase-exercise` branch.
  - Switch to the `main` branch and make additional changes.
  - Rebase the `feature/rebase-example` branch onto the `main` branch using `git rebase main`.
  - Check the commit history with `git log` to see the updated order of commits.

## 11.1 Stashing changes for later use

`git stash` is a valuable command in Git that allows you to save your current changes temporarily without committing them. This is useful when you need to switch branches or work on something

else without creating a commit for unfinished work. For example, when you're working on a feature and you need to switch to a different task quickly or your work gets interrupted unexpectedly, you may not want to commit your unfinished changes. When you run `git stash`, Git stores the changes in your working directory and staged changes in a special stash commit. The working directory is then reverted to the last committed state, providing a clean slate for your next task.

### 11.1.1 Stashing changes

To demonstrate the usefulness of `git stash`, you can edit one or more files in your repository, stage or don't stage your changes, but don't commit them. Now use `git status` to look at your working directory.

```
git status
```

Yo should get an output similar to:

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   example.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   example.txt
```

In this example, there are changes on two tracked files, one staged and one not staged, that we want to stash. Just like when committing, it is possible to add a message when stashing using the `-m` flag. It makes sense to add a message because it serves as a reminder of what changes were stashed, making it easier to identify the purpose of the stash when you later list or apply stashes.

```
git stash -m "feature X WIP"
```

You should get an output similar to:

```
Saved working directory and index state On feature: "feature X WIP"
```

You can now use `git status` to look at your working directory again and should see an output similar to:

```
On branch main
nothing to commit, working directory clean
```

Now you can switch branches and work on something else or stop working on the project altogether.

### 11.1.2 Retrieving stashed changes

If you want to reapply your changes, you can use `git stash apply`. To look at all your stored stashes, you can use `git stash list`.

```
git stash list
```

You should get an output similar to:

```
stash@{0}: On feature: feature X WIP
```

All of your stored stashes should show up, with the most recent on top. Every stash should have a number, applied chronologically. `git stash apply` automatically applies your latest stash. You can also specify a stored stash, for example `git stash apply stash@{3}`

```
git stash apply stash@{3}
```

You should get an output similar to:

```
On branch main
Changes not staged for commit:
  (use "git add <file>.. ." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   example.html
    modified:   example.txt

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

The changes made in your stored stashed are now applied again in your working directory. Note that already staged files are not automatically staged again.

> Common `git stash` flags
>
> `git stash -m "message"`: Save your changes in a stash with a message.
> `git stash apply stash@{n}`: Apply the changes from the specified stash (identified by its index `n`) to your working directory.
> `git stash pop stash@{n}`: Apply the changes from the specified stash (identified by its index `n`) to your working directory and remove the stash from the stash list.
> `git stash list`: List all the stashes you have created, showing their reference numbers and stash messages.
> `git stash drop stash@{n}`: Remove the specified stash (identified by its index `n`) from the stash list.
> `git stash branch <branchname>`: Create a new branch named `<branchname>` from the commit where you originally stashed your changes and apply the stash to the new branch.
> `git stash show stash@{n}`: Show the diff of the changes stored in the specified stash (identified by its index `n`).

## 11.2 Removing changes or files

> ⚠ Be careful about data loss!
>
> This section introduces Git commands that may delete changes in your repository beyond recovery. Especially if you apply them for the first time, take your time and only execute a command if you are certain about its consequences. Better yet, experiment with these commands in a separate test repository, not in a repository where you keep your most important work.

### 11.2.1 Discarding changes in the working directory

Sometimes you might want to revert files back to the state of your last commit. For example, this can be useful when you realize that the recent changes you made to a file have introduced an error. You can use `git restore` to **discard changes in the working directory** and revert the files to the state they were in when you last committed them.

```
git restore <file>
```

> ⚠ Be careful about data loss!
>
> Please note that it may not be possible to undo the restore, since Git does not keep track of the state before the restore operation!

It is also possible to **revert all files in your repository** to the state of your last commit using:

```
git restore .
```

### 11.2.2 Unstaging files

You can also use `git restore` with the `--staged` flag to unstage files, you have previously added to the staging area. This can be useful when you accidentally staged files or want to reorganize your commit.

```
git restore --staged <file>
```

Or:

```
git restore --staged .
```

### 11.2.3 Deleting files

If you want to delete files from your computer and your Git repository, you can use the `git rm` command. This command removes files from your working directory and automatically stages this change for your next commit. The workflow would go like this:

```
git rm <FILE>
```

```
git commit -m "Delete file <FILE>"
```

> **Common `git rm` flags**
>
> `-f` or `--force`: This flag forces the removal of files, even if they are modified.
> `-r` or `--recursive`: Use this flag to remove directories and their contents.
> `--cached`: This flag removes files from the staging area but preserves them in the working directory.
> `-n` or `--dry-run`: With this flag, Git will only show you what would be removed but will not actually perform the deletion.

### 11.2.4 "Untracking" files

Sometimes, you may have files, in your Git repository that you no longer want to track or include in future commits, but want to keep in your local filesystem. It makes sense to include these files in a `.gitignore` file, as discussed in the chapter on first steps with Git. You could also use `git rm` in combination with the `--cached` flag.

```
git rm --cached <file>
```

### 11.2.5 Reverting commits

The `git revert` command is used to create a **new commit that undoes the changes made in a previous commit**. It's a way to safely reverse the effects of a specific commit without actually removing that commit from the commit history. It requires specifying the commit hash you want to revert. To look at the hashes of your commits, you can use the `git log` command. If you have a specific commit that you want to revert, you would use a command like:

```
git revert <commithash>
```

This will create a new commit that effectively undoes the changes made in the specified commit. This new commit will have the opposite changes, effectively canceling out the changes from the original commit. You might need to resolve a merge conflict, if the changes you want to revert conflict with changes in subsequent commits. By default the commit message will be `revert <commit message of reverted commit>`. However, Git will open an editor for you to change this message, if you don't specify it otherwise.

**What are hashes?**

In Git, a commit hash, also known as a commit ID or SHA-1 hash, is a unique identifier for a specific commit in a Git repository. It's a 40-character hexadecimal string that represents the contents and history of that commit. Each commit in a Git repository has a unique hash.

**Common `git revert` flags**

`-n` or `--no commit`: Prevents Git from automatically creating a new commit after reverting changes. It stages the changes, allowing you to make additional modifications or review them before committing.
`-m <parent-number>`: When dealing with a merge commit, this flag specifies which parent commit to use as the source for reverting. By default, Git uses the first parent (main branch), but you can specify another parent by providing its number.
`--no-edit`: This flag prevents the text editor from opening for editing the commit message, making it useful when you want to keep the default commit message.

### 11.2.6 Resetting to a commit

If you want to reset your repository to a commit that was made in the past, you can use the `git reset` command.

To get an overview of your past commits you can either use `git log` or `git reflog`. Both commands should get you an overview of your past commits. For example:

```
git log --oneline
```

will give you an output like:

```
25a51e8 Update README.md
b7f3a12 Add new feature X
8d76a45 Merge branch 'feature-branch'
2f0e73b Implement new functionality
```

Using `git reset <commit>` will **undo all the changes you made after the specific commit** you picked. Your "branch pointer" will move to the specified commit, and changes after that commit will be uncommitted and moved back to the working directory. For example:

```
git reset 8d76a45
```

However, these changes are still present in your working directory, so if you want to discard them completely, you can use add the `--hard` flag:

```
git reset --hard <commit>
```

This will not only reset your “branch pointer” but also discard all changes in your working directory and staging area since the specified commit. It effectively resets your working directory to the state of the chosen commit.

> ⚠ Be careful about data loss!
>
> Please be careful when using the `git reset --hard` command! This command reset your working directory to the state of the chosen previous commit. This will delete changes beyond recovery, unless you can retrieve these changes from another location, for example a remote repository like GitHub.

> 💡 `git reflog` vs. `git log`
>
> `git log` shows the commit history, while `git reflog` shows a log of all changes to branches, including resets and other adjustments. Think of `git log` as a timeline of commits, and `git reflog` as a detailed diary of all recent changes, even those that alter commit history, providing a safety net for recovery.

> 💡 Common `git reset` flags
>
> `--soft`: Resets the branch pointer to the specified commit but keeps changes staged. This allows you to rework the changes and create a new commit.
> `--mixed` (default): Resets the branch pointer to the specified commit and unstages changes. Changes are kept in your working directory, allowing you to modify them before committing.
> `--hard`: Resets the branch pointer to the specified commit, unstages changes, and discards changes in your working directory. Use with caution, as it can lead to data loss.
> `--merge`: Resets the branch pointer and the index to the specified commit, but keeps changes in your working directory. This is useful in aborting a merge.
> `--keep`: Resets the branch pointer and index to the specified commit but refuses to do so if there are uncommitted changes in the working directory.
> `--patch` or `-p`: Allows you to interactively choose changes to reset, similar to the `git add -p` command for staging changes.

## 11.3 Partial commits

Git allows you to make partial commits by staging only specific parts of your file before committing. This can be achieved using `git add` with the `-p` or `--patch` option, which allows you to interactively choose which changes to stage.

To try this make some changes to your file(s) and use:

```
git add -p
```

This will prompt you with each change, giving you options to stage, skip, or split the changes. You’ll see a series of prompts like this:

```
+ Example text ...
+
+
+
(1/x) Stage this hunk [y,n,q,a,d,e,?]?
```

These prompt option respectively stand for:

- `y`: Stage this hunk.
- `n`: Do not stage this hunk.
- `q`: Quit. Do not stage this hunk or any remaining hunks.
- `a`: Stage this hunk and all later hunks in the file.
- `d`: Do not stage this hunk or any later hunks in the file.
- `/`: Search for a hunk matching the given regex.
- `e`: Manually edit the current hunk.
- `?`: Print help.

Type one of the symbols in the command line to proceed in the desired manner.

> **What is a "hunk"?**
>
> In Git, a "hunk" refers to a distinct block of code changes within a file. It represents a cohesive set of added, modified, or deleted lines in a specific location. Git automatically divides changes into hunks to facilitate easier review, selective staging, and conflict resolution during version control operations.

> **Our recommendation: Use a GUI for partial commits**
>
> In our opinion, using partial commits on the command line is a bit of a hassle. This would be a good usecase for a Git GUI. To checkout how to do partial commit using GitKraken checkout the GUI chapter in this book.

## 11.4 Alternatives to standard merging

### 11.4.1 Cherrypicking

`git cherry-pick` allows you to apply the changes from a specific commit from one branch to another. This means, you can pick and apply specific commits to your current branch without merging the entire branch. This can be useful when you only want to bring in specific changes from another branch into your current branch, in contrast to merging all commits of a branch, as visualized in Figure 11.1. You will need the hash of the commit you want to "cherry-pick" and then use the command:

```
git cherry-pick <commithash>
```

Again, you might have to resolve merge conflicts. The default commit message will be:

```
Cherry-pick commit <commit-hash>
This commit was cherry-picked from <source-branch> at <source-commit-hash>
```

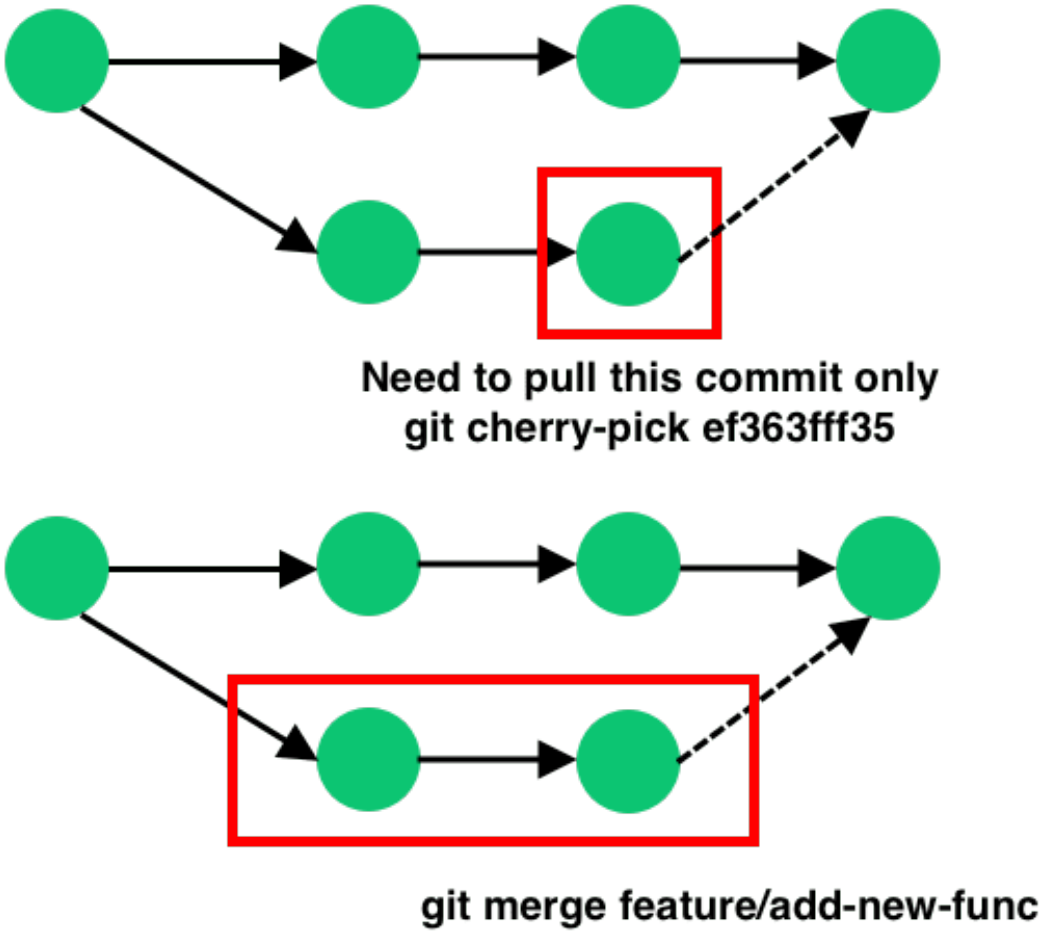



Figure 11.1: Image from Blogpost “What IS git cherry-pick?” by 0xkoji

**Common `git cherry-pick` flags**

`-n` or `--no commit`: Prevents Git from automatically creating a new commit after cherry-picking. It stages the changes, allowing you to make additional modifications or review them before committing.
`-e` or `--edit`: Opens the default text editor to edit the commit message of the new cherry-picked commit. Useful when you want to provide a custom message for the cherry-picked commit.

### 11.4.2 Rebasing

`git rebase` is a different way compared to a standard merge to integrate changes from one branch into another. For example, when you rebase the `feature` branch onto the `main` branch you “rear-

range" the commits. It's like taking your changes, applying them on top of the latest `main` branch, and making it all look like a smooth line. The new commits you made in the feature branch are still there, but they appear as if they were created after the latest changes in the main branch. It's like picking up your changes and placing them on the latest code, resulting in a linear history. For an illustration, see Figure 11.2.

```
git rebase main
```

However, you should use rebase with caution when collaborating with others, as it can rewrite commit history and create conflicts for team members.

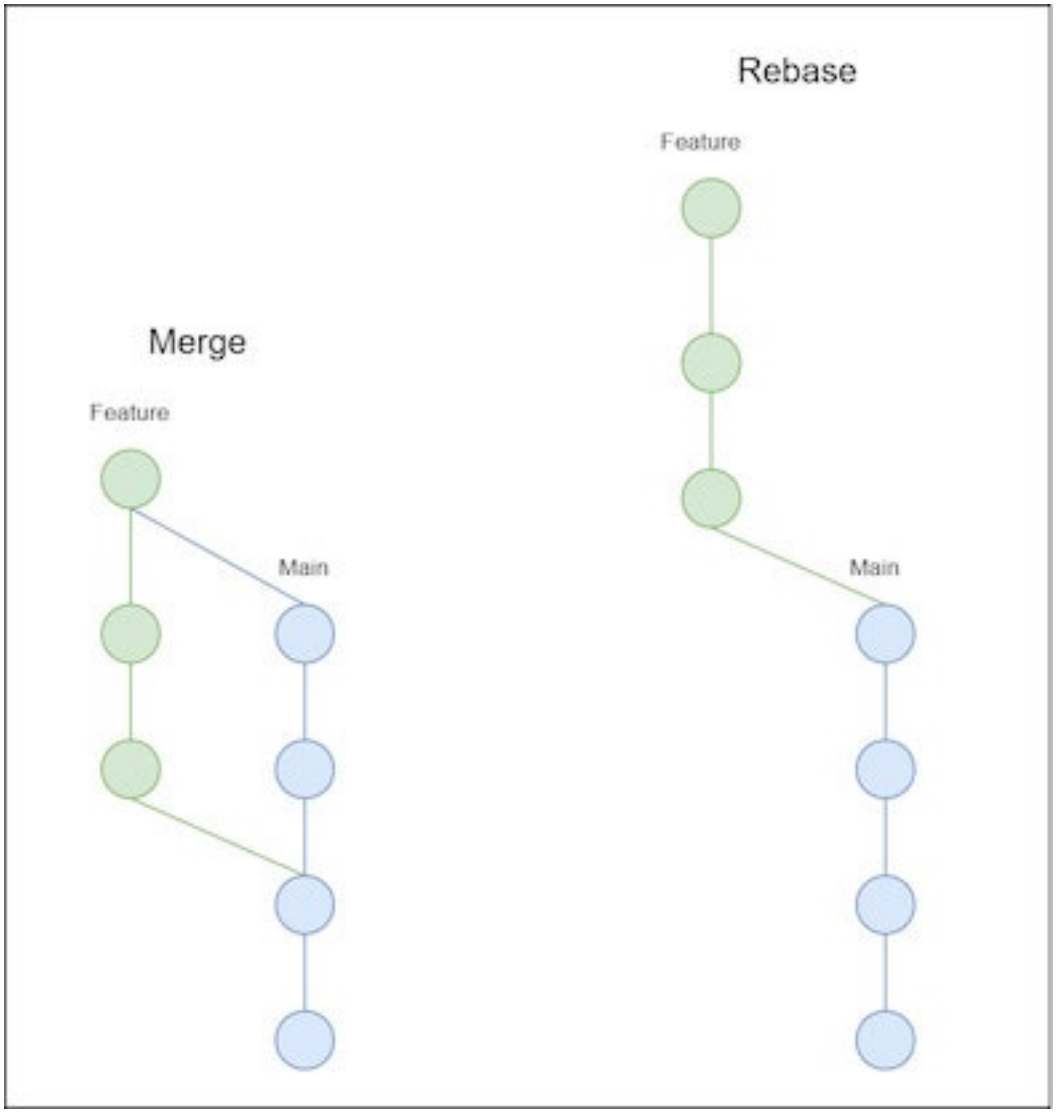



Figure 11.2: Image from Blogpost "Deep dive into Git Rebase" by Mohan Murali

Common `git rebase` flags

`--interactive` or `-i`: Start an interactive rebase, allowing you to edit, reorder, or squash commits interactively.

`--continue`: Continue the rebase after resolving conflicts or editing commits during an interactive rebase.
`--abort`: Abort the current rebase operation and return the branch to its original state before the rebase.
`--skip`: Skip the current commit during an interactive rebase.
`-p` or `--preserve-merges`: Preserve merge commits during the rebase.
`--autosquash`: Automatically squash commits marked with "squash" or "fixup" in their commit message during an interactive rebase.

## 11.5 Acknowledgements & further reading

We would like to express our gratitude to the following resources, which have been essential in shaping this chapter. We recommend these references for further reading:

| Authors | Title | Website | License | Source |
|---|---|---|---|---|
| Chacon and Straub [7] | Pro Git | | CC BY-NC | |

## 11.6 Cheatsheet

| Command | Description |
|---|---|
| `git rebase` | Different way of integrating changes from two branches |
| `git stash` | Stores made changes for later use |
| `git stash -m stashing message` | Stashes your changes and includes a message |
| `git stash list` | Shows all of your stored stashes |
| `git stash apply` | Applies your latest stash |
| `git stash apply stash@{n}` | Applies a specific stash |
| `git stash pop` | Applies your latest stash and removes it from stash list |
| `git stash pop stash@{n}` | Applies a specific stash and removes it from stash list |
| `git restore <file>` | Reverts back to the state of your last commit |
| `git restore .` | Reverts all files in your repository back to the state of your last commit |
| `git restore --staged <file>` | Removes from your staging area |
| `git restore --staged .` | Removes all files in your repository from your staging area |
| `git rm <file>` | Deletes from your repository |
| `git rm --cached <file>` | Removes from your repository but keeps on your system |
| `git revert <commithash>` | Creates a new commit which reverts your repository back to |

| Command | Description |
|---|---|
| `git reflog` | Logs recent branch changes |
| `git reset <commithash>` | Resets the branch to a specified commit, keeping changes in the working directory |
| `git reset --hard <commithash>` | Resets the branch to a specified commit |
| `git cherry-pick <commithash>` | Applies changes from |
| `git clean` | Deletes untracked files from your directory |

# 12 Rewriting History

Dive into Git's history-rewriting features for cleaner commits and efficient collaboration.

Learning Objectives
 Understanding the Risks of Sensitive Data in Git History  Mastering Tools for History Rewriting  Ensuring Collaborative Clean-Up

## 12.1 How to avoid accidental commits

Considering rewriting Git history? This usually comes into play when there are errors or changes needed in the initial commits. There are a few tricks to avoid committing things you don't want committed:

1. Avoid the catch-all commands `git add .` and `git commit -a`. Use `git add filename` and `git rm filename` to individually stage files.

2. Use `git add --interactive` to individually review and stage changes within each file.

3. Use git `diff --cached` to review the changes that you have staged for commit. This is the exact diff that `git commit` will produce as long as you don't use the `-a` flag.

4. You can use also a visual program like GitHub Desktop or GitKraken to commit changes. Visual programs generally make it easier to see exactly which files will be added, deleted, and modified with each commit.

## 12.2 Purging a file from your repository's history

Let's say that you accidentally added a large file to a previous commit. Now you want to remove the file but keep all commits that came afterwards.

For example, this could be the case if you committed sensitive data as a binary file. You will need to remove the file from the history, as you can't modify it to remove or replace the data.

### 12.2.1 git filter-repo

git filter-repo is a very useful tool that is not part of the official Git distribution. It is a third-party tool and is designed to be used alongside Git for more advanced repository history manipulation tasks.

To illustrate how `git filter-repo` works, we'll show you how to remove your file with sensitive data from the history of your repository and add it to `.gitignore` to ensure that it is not accidentally re-committed.

#### 12.2.1.1 Installation

Since `git filter-repo` is not a part of standard Git, you will need to install it first. You can install `git-filter-repo` manually or by using a package manager. For example, to install the tool with HomeBrew on macOS, use the `brew install` command.

```
brew install git-filter-repo
```

For more information on installation, see INSTALL.md in the newren/git-filter-repo.

#### 12.2.1.2 Clone a fresh copy of the repository

If you don't already have a local copy of your repository with sensitive data in its history, clone the repository to your local computer.

```
git clone https://github.com/YOUR-USERNAME>/YOUR-REPOSITORY ①
```

① Replace `YOUR-USERNAME` with your GitHub username and `YOUR-REPOSITORY` with the name of your GitHub repository.

#### 12.2.1.3 Navigate into the repository's working directory

Navigate into the repository's working directory. For details on folder navigation in the command line, see the "Command Line" chapter.

```
cd YOUR-REPOSITORY ①
```

① Replace `YOUR-REPOSITORY` with the name of your GitHub repository. For details on the `cd` command, see the "Command Line" chapter.

#### 12.2.1.4 Running `git filter-repo` command

> **Warning**
>
> If you run `git filter-repo` after stashing changes, you won't be able to retrieve your changes with other stash commands. Before running `git filter-repo`, we recommend unstashing any changes you've made. To unstash the last set of changes you've stashed, run `git stash show -p | git apply -R`. For more information, see Git Tools - Stashing and Cleaning.

Let's say that you accidentally committed (and pushed) a file you actually don't want to share because it contains sensitive data. After you installed the `git filter-repo` tool, you can use it to exclude a specific file from you commit history.

Run the following command, replacing `PATH-TO-FILE-YOU-WANT-TO-REMOVE` with the path to the file you want to remove, not just its file name. These arguments will do the following:

1. Force Git to process, but not check out, the entire history of every branch and tag
2. Remove the specified file, as well as any empty commits generated as a result 1- Remove some configurations, such as the remote URL, stored in the `.git/config` file. You may want to back up this file in advance for restoration later.
3. Overwrite your existing tags

Essentially, `git filter-repo` will modify your entire Git repository history to exclude the specified file (`PATH-TO-FILE-YOU-WANT-TO-REMOVE`). The `--invert-paths` flag inverts the selection, so it keeps everything except the specified path, effectively removing the file from the entire history of the repository. If you do not use `--invert-paths`, the command would only keep the specified file in your repository, discarding everything else.

```
git filter-repo --invert-paths --path PATH-TO-FILE-YOU-WANT-TO-REMOVE #<1>
```

① Replace `PATH-TO-FILE-YOU-WANT-TO-REMOVE` with the path to the file** you want to remove, not just its file name.

> **I did not clone a fresh copy of the repository. Is this a problem?**
>
> If you did not clone a fresh copy of your repository, you may see this message after running the `git filter-repo` command:
>
> ```
> Aborting: Refusing to destructively overwrite repo history since
> this does not look like a fresh clone.
>   (expected freshly packed repo)
> Please operate on a fresh clone instead.  If you want to proceed
> anyway, use --force.
> ```
>
> As described in the message, you have two options:
>
> 1. Clone a fresh copy of your repository and execute the command there.
> 2. Add `--force` to proceed with the existing repository.

#### 12.2.1.5 Add your file to `.gitignore`

Add your file to `.gitignore` to ensure that you don't accidentally commit it again. You can edit `.gitignore` in your favorite text editor.

```
echo "PATH-TO-FILE-YOU-WANT-TO-REMOVE" >> .gitignore                    ①
git add .gitignore                                                      ②
git commit -m "Add YOUR-FILE-WITH-SENSITIVE-DATA to .gitignore"         ③
```

① This command writes `PATH-TO-FILE-YOU-WANT-TO-REMOVE` inside the `.gitignore` file. `>>` makes sure that this is written to a new line inside the `.gitignore` file.

Double-check that you've removed everything you wanted to from your repository's history, and that all of your branches are checked out.

### 12.2.2 BFG

Instead of using `git filter-repo` it is also possible to use the open source tool BFG repo cleaner. The tool is especially useful if you need to remove very large files from your commit history.

### 12.2.3 Force-push your local changes to GitHub

If you are happy with the state of your repository, force-push your local changes to overwrite your repository to GitHub, as well as all the branches you've pushed up. A force push is required to remove sensitive data from your commit history.

When you perform a force push, you are essentially overwriting the existing commit history on the remote branch with the new commit history from your local branch. So instead of adding changes to your commit history, like with a normal push, a force push changes or discards the existing commit history on that branch. To do a force push, you use `git push` together with the `--force` flag. The `--all` flag is used to push all branches that have corresponding branches on the remote repository, that way you are updating multiple branches at once.

```
git push origin --force --all
```

Error: `fatal: 'origin' does not appear to be a git repository`

Did you receive this error message?

```
fatal: 'origin' does not appear to be a git repository
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

As described above, this could mean that `origin` was removed from your repository. Configure origin again using either SSH or HTTPS:

#### 12.2.4 SSH

```
git remote add origin https://github.com/YOUR-USERNAME/YOUR-REPOSITORY    ①
```

① Replace `YOUR-USERNAME` with your GitHub username and `YOUR-REPOSITORY` with the name of your GitHub repository.

#### 12.2.5 HTTPS

```
git remote add origin git@github.com:YOUR-USERNAME/YOUR-REPOSITORY.git    ①
```

① Replace `<YOUR-USERNAME>` with your GitHub username and `<YOUR-REPOSITORY>` with the name of your GitHub repository.

In order to remove the sensitive file from your tagged releases, you'll also need to force-push against your Git tags:

```
git push origin --force --tags
```

⚠ The dangers of force pushing

As stated in this example, force pushing can be very useful, for example if you committed and pushed data you want to remove again. However force pushing comes with potential drawbacks and should mostly be used as a last resort.
Force pushing can lead to the **loss of commit history** on the remote branch, which may cause confusion or conflicts for collaborators who have based their work on the previous history. Force pushing to a shared branch can **cause conflicts for collaborators** who have already cloned the repository and pulled changes. Once a force push is executed, the old **commit history may become difficult to recover.**
Especially if you collaborate with others on your repository, **let them know in advance** that you are force-pushing to your common remote repository.

## 12.3 Fully removing data from GitHub

If you you committed **really** sensitive data and want to make **absolutely** sure that it's deleted, you will have to complete a few additional steps. After using either the BFG tool or git filter-repo to remove the sensitive data and pushing your changes to GitHub, you must take a few more steps to fully remove the data from GitHub.

1. Contact GitHub Support, asking them to remove cached views and references to the sensitive data in pull requests on GitHub. Please provide the name of the repository and/or a link to the commit you need removed.

2. Tell your collaborators to rebase, not merge, any branches they created off of your old (tainted) repository history. One merge commit could reintroduce some or all of the tainted history that you just went to the trouble of purging.
3. After some time has passed and you're confident that BFG / git filter-repo had no unintended side effects, you can force all objects in your local repository to be dereferenced and garbage collected with the following commands (using Git 1.8.5 or newer):

```
git for-each-ref --format="delete %(refname)" refs/original | git update-ref --stdin
```

```
git reflog expire --expire=now --all
```

```
git gc --prune=now
```

> **Tip**
>
> You can also achieve this by pushing your filtered history to a new or empty repo.

## 12.4 Acknowledgements & further reading

We would like to express our gratitude to the following resources, which have been essential in shaping this chapter. We recommend these references for further reading:

| Authors | Title | Website | License | Source |
|---|---|---|---|---|
| GitHub [9] | GitHub Docs | | CC BY-NC 4.0 | |

## 12.5 Cheatsheet

| Command | Description |
|---|---|
| `git filter-repo --invert-paths --path < PATH-TO-FILE-YOU-WANT-TO-REMOVE >` | Remove specified file from your repository history |
| `brew install git-filter-repo` | Installs git filter-repo using brew |

# Exercises

## First steps with Git

To learn Git effectively, it makes sense to practice version control by implementing it on your own small project. For the purpose of this guide, we will start off with a small project that only involves plain-text files. You don't need to know any programming language like R or Python. So while reading this chapter and using the commands along, **your task** is to:

☐ Create a folder called `recipes` and initialize it as a Git repository. ☐ Create a plain text file called `recipes.txt` in this folder. ☐ **Add a recipe** in this file.

This can be the recipe for your favorite dish or an unusual AI-generated recipe. ☐ Stage and commit your changes to the file

Please **keep this project directory**! This guide will continue to use your recipe project as an example in following chapters.

There are several terms in these instructions that might be unfamiliar to you, for example "repository", "stage" or "commit". Don't worry, you will learn about what these terms mean in this chapter. Let's *git* started!

## Branches

☐ Create a new branch called `feature/newrecipe` ☐ Add a new recipe to your recipe text file
☐ Merge this branch with your `main` branch and delete the `feature` branch afterwards

## GitHub

☐ Upload your repository to GitHub

- Create a new repository on GitHub.
- Set the remote URL to your GitHub repository.
- Push your changes to GitHub.

☐ Clone your repository, push and pull

- Clone your uploaded repository to a different location on your computer.
- Make changes and push them to GitHub.
- Pull the changes into your original repository.

☐ Bonus: Create a Pull Request

*Exercises*

- Create a new branch in your local repository.
- Add, commit, and push changes in the new branch.
- Create and merge a pull request for this new branch on GitHub

## Tags and Releases

☐ Create a Lightweight Tag ☐ Create an Annotated Tag ☐ Switch to a Tag ☐ Delete a Tag ☐ Push a Tag to remote ☐ Create a GitHub release ☐ Link a Zenodo record

## Graphical User Interfaces

☐ Install GitKraken or GitHub Desktop ☐ Login to the client using your GitHub Account and view your recipe repository ☐ Add two recipes in the same file, stage and commit only one of them, using a GUI ☐ Open your recipe repository in RStudio, edit a file and commit your changes

## Stashing and Co.

☐ Stashing

- Create a new branch called `feature/stash-exercise`.
- Make changes to two different files in your project directory.
- Stash your changes without adding a message.
- Use `git status` to verify that your working directory is clean.
- Apply the stash to your working directory and verify that your changes are restored.

☐ Reverting

- Create a new branch called `feature/revert-exercise`.
- Make changes and commit these to your file.
- Use `git revert` to revert the most recent commit, specifying its hash.

☐ Rebasing

- Create a new branch called `feature/rebase-exercise`.
- Make three commits with minor changes on the `feature/rebase-exercise` branch.
- Switch to the `main` branch and make additional changes.
- Rebase the `feature/rebase-example` branch onto the `main` branch using `git rebase main`.
- Check the commit history with `git log` to see the updated order of commits.

# Cheatsheet

| Chapter | Command | Description |
|---|---|---|
| cli | `pwd` | Display the current directory path |
| cli | `cd <PATH>` | Change the current working directory to `<PATH>` |
| cli | `cd ~` | Change the current working directory to the user's home directory |
| cli | `cd ..` | Move up one folder |
| cli | `cd ../..` | Move up two folders |
| cli | `clear` | Clears content of of your terminal window |
| cli | `ls` | List files and folders in the current working directory |
| cli | `ls <PATH>` | List files and folders in `<PATH>` |
| cli | `ls -a` | List all files (including hidden files) in the current working directory |
| cli | `ls -alh` | List all files in a long format that is easy to read for humans |
| cli | `[Command] --help` | Provides all possible flags for a specific command (on Windows) |
| cli | `man [Command]` | Provides all possible flags for a specific command (on Mac OS) |
| cli | `mkdir <FOLDER>` | Create a new folder, called `<FOLDER>` |
| cli | `mkdir <FOLDER1> <FOLDER2>` | Create two separate folders, called and |
| cli | `touch <FILE>` | Create a new empty file, called `<FILE>` |
| cli | `open <FILE>` | Opens the file called (On macOS) |
| cli | `echo 'example text' >> file.txt` | Writes 'example text' into `file.txt` |
| cli | `cat <FILE>` | Displays the content of |
| cli | `start <FILE>` | Opens the file called (On Windows) |
| cli | `mv FILE.txt <FOLDER>` | Move `FILE.txt` into `<FOLDER>` |
| cli | `mv <FOLDER_OLD> <FOLDER_NEW>` | Rename `<FOLDER_OLD>` to `<FOLDER_NEW>` |
| cli | `ls -alh *.csv` | Use a wildcard to list all `.csv` files |
| cli | `rm -r <FOLDER>` | Remove the folder `<FOLDER>` |
| config | `git config` | Overview of config commands |
| config | `git config --global user.name` | Set Username |
| config | `git config --global user.email` | Set Email |
| config | `git config --global core.editor` | Set Editor |
| config | `git config --global init.defaultBranch main` | Set default branch name |

| Chapter | Command | Description |
|---|---|---|
| config | `git config --list` | View set configurations |
| basic | `git init` | Initialize folder as Git Repository |
| basic | `git status` | View Git tracking status of files in Repository |
| basic | `git add` | Add file to staging area |
| basic | `git commit` | Commit staged files |
| basic | `git log` | View past commits |
| basic | `git diff` | View made changes compared to the last commit |
| github | `git clone` | Create a local copy of a repository |
| github | `git pull` | Fetches and merges the latest changes from a remote repository into your current branch |
| github | `git fetch` | Updates your remote tracking branches |
| github | `git push` | Upload your local commits to a remote repository |
| github | `git blame` | shows the authorship and commit information of each line in a file |
| github | `git remote` | manages remote repositories |
| branches | `git branch` | Listing/creating/deleting branches |
| branches | `git switch` | Switching branches |
| branches | `git checkout` | Switching branches |
| branches | `git merge` | Merging branches |
| branches | `git rebase` | A different way of integrating changes from two branches |
| stashing | `git rebase` | Different way of integrating changes from two branches |
| stashing | `git stash` | Stores made changes for later use |
| stashing | `git stash -m stashing message` | Stashes your changes and includes a message |
| stashing | `git stash list` | Shows all of your stored stashes |
| stashing | `git stash apply` | Applies your latest stash |
| stashing | `git stash apply stash@{n}` | Applies a specific stash |
| stashing | `git stash pop` | Applies your latest stash and removes it from stash list |
| stashing | `git stash pop stash@{n}` | Applies a specific stash and removes it from stash list |
| stashing | `git restore <file>` | Reverts back to the state of your last commit |
| stashing | `git restore .` | Reverts all files in your repository back to the state of your last commit |
| stashing | `git restore --staged <file>` | Removes from your staging area |
| stashing | `git restore --staged .` | Removes all files in your repository from your staging area |
| stashing | `git rm <file>` | Deletes from your repository |
| stashing | `git rm --cached <file>` | Removes from your repository but keeps on your system |
| stashing | `git revert <commithash>` | Creates a new commit which reverts your repository back to |
| stashing | `git reflog` | Logs recent branch changes |

| Chapter | Command | Description |
|---|---|---|
| stashing | `git reset <commithash>` | Resets the branch to a specified commit, keeping changes in the working directory |
| stashing | `git reset --hard <commithash>` | Resets the branch to a specified commit |
| stashing | `git cherry-pick <commithash>` | Applies changes from |
| stashing | `git clean` | Deletes untracked files from your directory |
| Tags and Releases | `git tag` | Lists all tags |
| Tags and Releases | `git tag v1.0` | Creates a tag on the basis of your current commit named `v1.0` |
| Tags and Releases | `git tag v1.1 <commit-hash>` | Creates a tag on the basis of a specific commit hash named `v1.1` |
| Tags and Releases | `git tag -a v1.0 -m Release version 1.0` | Creates an annotated tag on the basis of your current commit hash named `v1.0` with the tagging message `Release version 1.0` |
| Tags and Releases | `git push origin <tag-name>` | Pushes a specific tag to remote |
| Tags and Releases | `git push origin --tags` | Pushes all created tags to remote |
| Tags and Releases | `git fetch --tags` | Fetches all created tags from remote |
| Tags and Releases | `git pull --tags` | Pulls all created tags from remote |
| rewriting_history | `git filter-repo --invert-paths --path < PATH-TO-FILE-YOU-WANT-TO-REMOVE >` | Remove specified file from your repository history |
| rewriting_history | `brew install git-filter-repo` | Installs git filter-repo using brew |

# References


[1] AI for Multiple Long-term Conditions Research Support Facility. *Introduction to version control with git*. License: CC BY 4.0. Source: https://github.com/aim-rsf/training/tree/main/version-control. June 2023. URL: https://www.youtube.com/watch?v=z9-qAGq78qE.

[2] J.J. Allaire et al. *Quarto*. Version 1.2. License: GNU GPL v2. https://github.com/quarto-dev/quarto-web. Website: https://quarto.org/. Jan. 10, 2022. DOI: 10.5281/zenodo.5960048. URL: https://github.com/quarto-dev/quarto-cli.

[3] John D. Blischak, Emily R. Davenport, and Greg Wilson. "A Quick Introduction to Version Control with Git and GitHub". In: *PLOS Computational Biology* 12.1 (Jan. 19, 2016). Ed. by Francis Ouellette. License: CC BY 4.0. Source: https://doi.org/10.1371/journal.pcbi.1004668, e1004668. DOI: 10.1371/journal.pcbi.1004668. URL: http://dx.doi.org/10.1371/journal.pcbi.1004668.

[4] Jennifer Bryan. "Excuse Me, Do You Have a Moment to Talk About Version Control?" In: *The American Statistician* 72.1 (Jan. 2, 2018). Website: https://doi.org/10.1080/00031305.2017.1399928., pp. 20–27. DOI: 10.1080/00031305.2017.1399928. URL: http://dx.doi.org/10.1080/00031305.2017.1399928.

[5] Jenny Bryan. *Happy Git and GitHub for the useR*. License: CC BY-NC 4.0. Source: https://github.com/jennybc/happy-git-with-r. Website: https://happygitwithr.com/. Jenny Bryan, 2023. URL: https://happygitwithr.com.

[6] Gerard Capes et al. *swcarpentry/shell-novice: Software Carpentry: the UNIX shell*. License: CC BY 4.0. Source: https://github.com/swcarpentry/shell-novice. Website: https://swcarpentry.github.io/shell-novice/. Zenodo, May 2, 2023. DOI: 10.5281/ZENODO.595899. URL: https://zenodo.org/record/595899.

[7] Scott Chacon and Ben Straub. *Pro Git*. License: CC BY-NC. Source: https://github.com/progit/progit2. Website: https://git-scm.com/book/en/v2. Apress, 2014. DOI: 10.1007/978-1-4842-0076-6. URL: http://dx.doi.org/10.1007/978-1-4842-0076-6.

[8] coderefinery. *GitHub without the command line*. License: CC BY-NC 4.0. Source: https://github.com/coderefinery/github-without-command-line/tree/master. Website: https://coderefinery.github.io/github-without-command-line/. coderefinery, 2023. URL: https://coderefinery.github.io/github-without-command-line/.

[9] GitHub. *GitHub Docs*. License: CC BY-NC 4.0. Source: https://docs.github.com/en. GitHub, 2023. URL: https://docs.github.com/en.

[10] Katherine E. Koziar et al. *swcarpentry/git-novice: Software Carpentry: Version Control with Git 2023-05*. License: CC BY 4.0. Source: https://github.com/swcarpentry/git-novice. Website: https://swcarpentry.github.io/git-novice/. Zenodo, May 8, 2023. DOI: 10.5281/ZENODO.7908089. URL: https://zenodo.org/record/7908089.

[11] Julia Stewart Lowndes and Erin Robinson. “Openscapes Champions Lesson Series”. In: (Dec. 6, 2022). License: CC BY 4.0. Source: https://github.com/Openscapes/series. Website: https://openscapes.github.io/series/. DOI: 10.5281/ZENODO.7407247. URL: https://zenodo.org/record/7407247.

[12] Miles McBain. *Git for Scientists*. License: CC BY-SA 4.0. Source: https://github.com/MilesMcBain/git_4_sci/. Website: https://milesmcbain.github.io/git_4_sci/. 2019. URL: https://milesmcbain.github.io/git_4_sci/.

[13] Ian Milligan and James Baker. “Introduction to the Bash Command Line”. In: *Programming Historian* 3 (Sept. 20, 2014). Ed. by Adam Crymble. DOI: 10.46430/phen0037. URL: https://dx.doi.org/10.46430/phen0037.

[14] K. Jarrod Millman et al. “Teaching Computational Reproducibility for Neuroimaging”. In: *Frontiers in Neuroscience* 12 (Oct. 22, 2018). License: CC BY 4.0. Website: http://dx.doi.org/10.3389/fnins.2018.00727. DOI: 10.3389/fnins.2018.00727. URL: https://dx.doi.org/10.3389/fnins.2018.00727.

[15] Yasset Perez-Riverol et al. “Ten Simple Rules for Taking Advantage of Git and GitHub”. In: *PLOS Computational Biology* 12.7 (July 14, 2016). Ed. by Scott Markel. License: CC BY 4.0. Source: https://github.com/ypriverol/github-paper. Website: https://doi.org/10.1371/journal.pcbi.1004947, e1004947. DOI: 10.1371/journal.pcbi.1004947. URL: http://dx.doi.org/10.1371/journal.pcbi.1004947.

[16] Karthik Ram. “Git can facilitate greater reproducibility and increased transparency in science”. In: *Source Code for Biology and Medicine* 8.1 (Feb. 28, 2013). License: CC BY 2.0. Source: https://github.com/karthik/smb_git. Website: https://doi.org/10.1186/1751-0473-8-7. DOI: 10.1186/1751-0473-8-7. URL: http://dx.doi.org/10.1186/1751-0473-8-7.

[17] The Turing Way Community. *The Turing Way: A handbook for reproducible, ethical and collaborative research*. License: The process documents and data are made available under a CC BY 4.0 license. Software are made available under an MIT license. Website: https://the-turing-way.netlify.app/. Zenodo, July 27, 2022. DOI: 10.5281/zenodo.3233853. URL: https://zenodo.org/record/3233853.

# Acknowledgements


### Funding

This work is funded by the Digital and Data Literacy in Teaching Lab (DDLitLab), an initiative by the Center for Interdisciplinary Study Programs (Zentrum für Interdisziplinäre Studienangebote; ISA-Zentrum) at the University of Hamburg, Germany. The Digital and Data Literacy in Teaching Lab program is in turn funded by the Stiftung Innovation in der Hochschullehre (details about the funding program can be found here).

Our project proposal can be found here (in German).

Special thanks to Carolin Scharfenberg as well as Moritz Kreinsen and Sören-Kristian Berger for help with project coordination, administration, and the grant application process.


### Code

This website is made with □ and Quarto. The source code of this website can be found on GitHub .

### Images

Images are stored on Nextcloud (UHH Cloud). They can be retrieved using

```
make download-images
```

Click here to view the full `Makefile` of this project.

### Analytics

We use Plausible for website analytics. Plausible is an “intuitive, lightweight and open source web analytics”, does not use cookies and is fully compliant with GDPR, CCPA and PECR.

### License

**Listing 12.1** `Makefile`

```
# define URL to Nextcloud where static files are stored:
IMAGES_URL=https://cloud.uni-hamburg.de/s/aD7NTNB9f4NDorT/download
# define a name for the .zip-archive with the cloud contents:
IMAGES_ARCHIVE=version-control-book.zip
# define the name of the local folder:
IMAGES_DIR=static/

# define the default targets of the make command
all: download-images

.PHONY: preview
preview:
    quarto preview --to html

.PHONY: html
html:
    quarto render --to html

.PHONY: pdf
pdf:
    quarto render --to pdf

# download and extract images:
download-images:
    wget $(IMAGES_URL) -O $(IMAGES_ARCHIVE)
    unzip -j -o $(IMAGES_ARCHIVE) -d $(IMAGES_DIR)
    rm -f $(IMAGES_ARCHIVE)

# clean downloaded images folder:
clean:
    rm -rf $(IMAGES_DIR)* _book/
```

## Contact

Did you spot an error? Is there an issue with the website? Do you have feedback, questions or comments? We would love to hear from you! Please open an issue on GitHub or send an email. Thank you!

# Contributing

## Overview

**?@sec-rstudio-project**: This section explains how to **use RStudio Project** to work on this project.

**?@sec-quarto**: This section explains how to **use Quarto** to work on this project.

**?@sec-dependency-management**: This section explains how this project **manages package dependencies** using `renv`.

**?@sec-code**: This section explains how to **add code snippets**.

**?@sec-images**: This section explains how to **add images**.

**?@sec-references**: This section explains how to **add references**.

**?@sec-variables**: This section explains how to **use variables**.

**?@sec-icons**: This section explains how to **add icons**.

## RStudio Project

- "Using RStudio Projects"

We recommend working in RStudio and open the RStudio Project.

## 12.6 RStudio

- Double-click on the project file `version-control-book.Rproj` within your computer's file system (for example, macOS Finder).

## 12.7 Terminal

In the Terminal, run:

```
open version-control-book.Rproj
```

This will open the RStudio Project in RStudio.

Click here to view the full `version-control-book.Rproj` file of this project.

**Listing 12.2** `version-control-book.Rproj`

```
Version: 1.0

RestoreWorkspace: No
SaveWorkspace: No
AlwaysSaveHistory: No

EnableCodeIndexing: Yes
UseSpacesForTab: Yes
NumSpacesForTab: 2
Encoding: UTF-8

RnwWeave: Sweave
LaTeX: pdfLaTeX

BuildType: Makefile
```

## Quarto

This book is made with Quarto.

### Local development

#### Setup

1. Read the Get Started chapter of the Quarto documentation, which includes details on installation.

#### Preparation

1. Read this guide on how to contribute.
2. Familiarize yourself with Quarto Books.

#### Local preview

```
quarto preview
```

### Add a new chapter

1. Create a new Quarto document inside the `/chapters/` folder
2. Add the new chapter to `_quarto.yml`. Don't forget to specify the full path (including `/chapters/`)

**Listing 12.3** `quarto.yml (excerpt)`

```
chapters:
  - index.qmd
  - chapters/YOUR-NEW-CHAPTER.qmd                                    ①
  - another-chapter.qmd
```

① Replace `YOUR-NEW-CHAPTER` with a concise filename.

Click here to view the full `_quarto.yml` file of this project.

## Dependency Management

We use `renv` for project-local R dependency management. For details on `renv`, see the renv documentation.

### Using `renv` for the first time

After you open the Rstudio Project, you will (hopefully) see a similar output in the RStudio console:

```
# Bootstrapping renv 0.17.3 ------------------------------------------------
* Downloading renv 0.17.3 ... OK (downloaded binary)
* Installing renv 0.17.3 ... Done!
* Successfully installed and loaded renv 0.17.3.
* Project '~/version-control-book' loaded. [renv 0.17.3]
* This project contains a lockfile, but none of the recorded packages are installed.
* Use `renv::restore()` to restore the project library.
```

This output tells you that (1) `renv` was automatically installed and loaded, (2) the `renv` environment for the project was loaded, and (3) that none of packages recorded in the `renv` lockfile are installed.

The `renv` lockfile captures the state of the package library of the project at some point in time. It is defining the **version of** `renv` used when generating the lockfile, the **version of** R used in that project, the **R repositories that were active** when the lockfile was created and the **package records** defining each R package, their version, and their installation source. For details on the anatomy of `renv.lock`, see this section in the `renv` documentation.

Click here to view the full `renv.lock` file of this project.

### Restoring `renv`

If `renv` is activated but it recognized that (not all of) the packages listed in the `renv.lock` file are installed, `renv` will suggest to restore the project library using `renv::restore()`:

```
* This project contains a lockfile, but none of the recorded packages are installed.
* Use `renv::restore()` to restore the project library.
```

```
renv::restore()
```

This will generate a list of files that will be updated.

Click here to view an example output of the `renv::restore()` command.

```
The following package(s) will be updated:

# CRAN ===============================
- R6             [* -> 2.5.1]
- base64enc      [* -> 0.1-3]
- bslib          [* -> 0.5.0]
- cachem         [* -> 1.0.8]
- cli            [* -> 3.6.1]
- data.table     [* -> 1.14.8]
- digest         [* -> 0.6.31]
- ellipsis       [* -> 0.3.2]
- evaluate       [* -> 0.21]
- fastmap        [* -> 1.1.1]
- fontawesome    [* -> 0.5.1]
- fs             [* -> 1.6.2]
- glue           [* -> 1.6.2]
- here           [* -> 1.0.1]
- highr          [* -> 0.10]
- htmltools      [* -> 0.5.5]
- jquerylib      [* -> 0.1.4]
- jsonlite       [* -> 1.8.5]
- knitr          [* -> 1.43]
- lifecycle      [* -> 1.0.3]
- magrittr       [* -> 2.0.3]
- memoise        [* -> 2.0.1]
- mime           [* -> 0.12]
- rappdirs       [* -> 0.3.3]
- rlang          [* -> 1.1.1]
- rmarkdown      [* -> 2.22]
- rprojroot      [* -> 2.0.3]
- sass           [* -> 0.4.6]
- stringi        [* -> 1.7.12]
- stringr        [* -> 1.5.0]
- tinytex        [* -> 0.45]
```

```
- vctrs          [* -> 0.6.3]
- xfun           [* -> 0.39]
- yaml           [* -> 2.3.7]

Do you want to proceed? [y/N]:
```

If you want to proceed with the update, type `y` and hit enter.

### Adding / removing packages

Install packages:

```
install.packages("PACKAGE_NAME")  ①
renv::install("PACKAGE_NAME")     ②
```

① Replace `PACKAGE_NAME` with the actual name of the package that you want to install.
② Replace `PACKAGE_NAME` with the actual name of the package that you want to install.

Remove packages:

```
remove.packages("PACKAGE_NAME")  ①
renv::remove("PACKAGE_NAME")     ②
```

① Replace `PACKAGE_NAME` with the actual name of the package that you want to remove.
② Replace `PACKAGE_NAME` with the actual name of the package that you want to remove.

Call `renv::snapshot()` to save the new state of the project library to the `renv.lock`:

```
renv::snapshot()
```

### Commit your changes

The `renv.lock` is checked into version control, so after you updated it, make a new commit:

```
git add renv.lock
git commit -m "add package PACKAGE_NAME to renv"  ①
```

① Replace `PACKAGE_NAME` with the actual name of the package that you installed. Adapt the commit message as needed.

## References

- "Citations & Footnotes" in the Quarto documentation.
- HTML options for references.

### Cite a reference

References are stored in `references.bib`.

Click here to view the full `references.bib` file of this project.

Use the citation handle of the `.bib`-entry that you want to cite. For example use `@chacon2014` to cite Chacon and Straub [7]. For details, see the chapter on "Citations & Footnotes" in the Quarto documentation. Add the specific references that you used to the bottom of the document. If the reference consists of multiple chapters and pages, try to specify the exact chapter or page.

### Add a new reference

1. Open `references.qmd`
2. Switch to the `Visual` Quarto editor.
3. Move the cursor into any place in the document.
4. Click on `Insert` and select `@ Citation ...`
5. If the reference has a DOI:
6. Select `From DOI`
7. Paste the DOI
8. Select `Insert`
9. This should add the new reference to `references.bib`.
10. Move the citation key to the correct place

### Add license information

We want to keep track of license information. If you add a new reference, please add the license information manually.

1. Go to `references.bib` and find the reference.
2. Inside the `.bib`-entry for the reference add a new field called `note` that includes the license information.

```
@book{
  note = {License: CC BY-NC}
}
```

For example, this is the references for Chacon and Straub [7]:

```
@book{chacon2014,
    title = {Pro Git},
    author = {Chacon, Scott and Straub, Ben},
    year = {2014},
    date = {2014},
    publisher = {Apress},
    doi = {10.1007/978-1-4842-0076-6},
    url = {http://dx.doi.org/10.1007/978-1-4842-0076-6},
```

```
    note = {License: CC BY-NC}
}
```

## Code

- "HTML Code Blocks" in the Quarto documentation

### Add code snippets

Add code snippets to a Quarto document like this:

````
```{bash}
git status
```
````

> **Disable code execution in all Quarto documents by default!**
>
> Place the following code in the YAML header of **each** Quarto document:
>
> ```
> engine: knitr
> execute:
>   eval: false
> ```
>
> Example: Click here for the YAML header of the current document.
>
> **Listing 12.7** `misc/contributing.qmd`
>
> ````
> 1 ---
> 2 author: ""
> 3 title-block-style: none
> 4 ---
> 5
> 6 ```{r}
> ````
>
> Click here to learn more.
> Quarto enables the inclusion of executable code blocks in Markdown. This empowers users to create reproducible documents and reports, as the code needed for generating the output is embedded within the document and automatically executed during rendering.
> However, in this guide, we usually only want to **display code but not execute it**! We therefore need to disable code execution in the YAML header of each Quarto document.
> The YAML header in a Quarto document is used to specify important metadata and settings for the document. It is typically placed at the top of the document enclosed between three dashes (---) to separate it from the main content. The YAML header provides instructions to the Quarto rendering engine on how to process and present the document.

### Reference external code

You can reference external code. For example, the following code block will:

| | description | code |
|---|---|---|
| 1 | reference the `_quarto.yml` file | `#| file: _quarto.yml` |
| 2 | display the first five lines of code | `echo=c(1:5)` |
| 3 | add code line numbers | `#| code-line-numbers: true` |
| 4 | add a filename to the code block | `filename="_quarto.yml"` |

````
```{bash, filename="_quarto.yml", echo=c(1:5)}
#| file: _quarto.yml
#| code-line-numbers: true
```
````

Click here for the output of this example.

### Add code annotations

- "Code Annotation" in the Quarto documentation

  Code blocks and executable code cells in Quarto can include line-based annotations. Line-based annotations provide a way to attach explanation to lines of code much like footnotes.

For example, this is a code annotation for the `git status` command:

````
```{bash}
git status                                                                    (1)
```
1. `git status` displays the state of the working directory and the staging area.
````

## Images

### Background

Images are stored in a NextCloud folder. They are downloaded into the `/static` folder using the command specified in the `Makefile`.

### Installation

You need to install the following packages:

1. GNU Wget
2. GNU Make

### 12.7.1

1. **Prerequisite:** Install Homebrew

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)
```

2. After you installed Homebrew, install GNU Wget

```
brew install wget
```

GNU Make should already be installed. Try by entering the following command into the Terminal:

```
make --version
```

### 12.7.2

TBA

### 12.7.3

TBA

#### Retrieve images

After you installed all the required software, run this command:

```
make download-images
```

What's a Makefile and what does it do?

**Listing 12.9** `Makefile`

```
# define URL to Nextcloud where static files are stored:
IMAGES_URL=https://cloud.uni-hamburg.de/s/aD7NTNB9f4NDorT/download
# define a name for the .zip-archive with the cloud contents:
IMAGES_ARCHIVE=version-control-book.zip
# define the name of the local folder:
IMAGES_DIR=static/

# define the default targets of the make command
all: download-images

.PHONY: preview
preview:
    quarto preview --to html

.PHONY: html
html:
    quarto render --to html

.PHONY: pdf
pdf:
    quarto render --to pdf

# download and extract images:
download-images:
    wget $(IMAGES_URL) -O $(IMAGES_ARCHIVE)
    unzip -j -o $(IMAGES_ARCHIVE) -d $(IMAGES_DIR)
    rm -f $(IMAGES_ARCHIVE)

# clean downloaded images folder:
clean:
    rm -rf $(IMAGES_DIR)* _book/
```

### Add a new image

1. Become a collaborator on the UHH Cloud folder.
2. Go to the UHH Cloud folder.
3. Place the new image inside the folder. Please use - to separate word in the filename.
4. Reference the image in the Quarto document relative to the root of the project directory:

```
![](static/NEW-IMAGE.png)
```

### Download images in pre-render command

It's also possible to include the `make download-images` as a project script in the `pre-render` script. This would re-execute `make download-images` before every render. For the moment, we disabled this option, but it can be enabled in `_quarto.yml`.

## Variables

- "Variables" in the Quarto documentation

Quarto offers a number of ways to **include dynamic variables within documents**. Dynamic variables are useful for externalizing content that varies depending on context, or as an alternative to repeating a value in multiple places.

In this project, we use a project-level `_variables.yml` file to define variables. Variables can be either simple values or can include arbitrary Markdown content.

Click here to view the full `_variables.yml` file of this project.

The `var` shortcode then allows to include references to those variables within any document in this project. For example, to include the link to the Nextcloud folder that stores static contents (images etc.) as a variable in a document, use `{{< var links.nextcloud >}}`.

For more details on variables, read the chapter on Variables in the Quarto documentation.

## Fontawesome Icons

- Font Awesome Extension for Quarto
- Free Font Awesome icons

We use the Font Awesome Extension for Quarto to include Font Awesome icons.

> This extension provides support including free icons provided by Font Awesome. Icons can be used in both HTML (via Font Awesome 6 Free) and PDF (via the fontawesome5 LaTeX package).

This extension includes support for **only free Font Awesome icons**.

To embed an icon, use the `{{< fa >}}` shortcode. For example, `{{< fa thumbs-up >}}` will result in . For details, see the project documentation on GitHub.

## Style Guide

We have compiled a set of guidelines to keep a consistent style across all chapters of the book.

### Summary

1. Write one sentence per line
2. Use consistent language

### Write one sentence per line

Please write each sentence on a new line. Having each sentence on a new line will make no difference to how the text is displayed, there will still be paragraphs, but it will mean that any pull requests will be easier to check; the changes will be on a single line instead of somewhere in a paragraph. Consider the example below.

```
Today you are you, that is truer than true. There is no one alive who is youer than you. - D
```

A pull request on this correcting it to have a '.' after Dr would show as a change to the whole paragraph. Contrast this with the next example which will be displayed online in the exact same way, but would see a change to a single line.

```
Today you are you, that is truer than true.
There is no one alive who is youer than you.
- Dr Seuss
```

### Use consistent language

We try to use consistent language:

| TRUE | FALSE |
|---|---|
| don't | do not |
| folder | directory |
| documentation | docs |

## Acknowledgements & Attribution

- The Quarto documentation [2], in particular the chapters on Variables, Quarto Extensions, "Code Annotation" (License: GNU GPL v2)
- [3]
- The Font Awesome Extension for Quarto (License: MIT)
- The Turing Way Community [17]: Style Guide

**Listing 12.4** `quarto.yml`

```yaml
project:
  # project options (https://quarto.org/docs/reference/projects/options.html):
  type: book
  output-dir: _book
  execute-dir: project
  # pre-render: make download-images
  # preview (https://quarto.org/docs/reference/projects/options.html#preview):
  preview:
    port: 3333
    host: 127.0.0.1
    browser: true
    watch-inputs: true
    navigate: true

book:
  title: "Version Control of Code and Data"
  subtitle: "Track, organize and share your work: An introduction to Git for research"
  author:
    - "Lennart Wittkuhn [{{< fa envelope >}}](mailto:lennart.wittkuhn@uni-hamburg.de) [{{< fa
    - "Konrad Pagenstedt [{{< fa envelope >}}](mailto:konrad@pagenstedt.de)"
  search: true
  repo-url: https://github.com/lnnrtwttkhn/version-control-book/
  repo-branch: main
  repo-actions: [edit, issue, source]
  # downloads: [pdf, epub]
  favicon: static/logo.svg
  twitter-card: false
  site-url: https://lennartwittkuhn.com/version-control-book
  chapters:
    - index.qmd
    - contents.qmd
    - misc/objectives.qmd
    - chapters/intro-version-control.qmd
    - chapters/command-line.qmd
    - chapters/installation.qmd
    - chapters/setup.qmd
    - chapters/first-steps-git.qmd
    - chapters/branches.qmd
    - chapters/github-intro.qmd
    - chapters/tags-and-releases.qmd
    # - chapters/project-management.qmd
    - chapters/issues.qmd
    - chapters/gui.qmd
    - chapters/intermediate-commands.qmd
    - chapters/rewriting-history.qmd
    - misc/exercises.qmd
    - misc/cheatsheet.qmd
    - misc/references.qmd
    - misc/acknowledgements.qmd
    - misc/contributing.qmd
  page-navigation: true
  back-to-top-navigation: true
  # side navigation (https://quarto.org/docs/websites/website-navigation.html#side-navigation)
  sidebar:
```

**Listing 12.5** `renv.lock`

```
{
  "R": {
    "Version": "4.3.1",
    "Repositories": [
      {
        "Name": "CRAN",
        "URL": "https://cran.rstudio.com"
      }
    ]
  },
  "Packages": {
    "R6": {
      "Package": "R6",
      "Version": "2.5.1",
      "Source": "Repository",
      "Repository": "CRAN",
      "Requirements": [
        "R"
      ],
      "Hash": "470851b6d5d0ac559e9d01bb352b4021"
    },
    "Rcpp": {
      "Package": "Rcpp",
      "Version": "1.0.10",
      "Source": "Repository",
      "Repository": "CRAN",
      "Requirements": [
        "methods",
        "utils"
      ],
      "Hash": "e749cae40fa9ef469b6050959517453c"
    },
    "askpass": {
      "Package": "askpass",
      "Version": "1.1",
      "Source": "Repository",
      "Repository": "CRAN",
      "Requirements": [
        "sys"
      ],
      "Hash": "e8a22846fff485f0be3770c2da758713"
    },
    "base64enc": {
      "Package": "base64enc",
      "Version": "0.1-3",
      "Source": "Repository",
      "Repository": "CRAN",
      "Requirements": [
        "R"
      ],
      "Hash": "543776ae6848fde2f48ff3816d0628bc"
    },
    "bib2df": {
      "Package": "bib2df",
```

**Listing 12.6** `references.bib`

```
@book{community2022,
    title = {The Turing Way: A handbook for reproducible, ethical and collaborative research},
    author = {{The Turing Way Community}},
    year = {2022},
    month = {07},
    date = {2022-07-27},
    publisher = {Zenodo},
    doi = {10.5281/zenodo.3233853},
    url = {https://zenodo.org/record/3233853},
    note = {License: The process documents and data are made available under a \href{https://c
}

@book{chacon2014,
    title = {Pro Git},
    author = {Chacon, Scott and Straub, Ben},
    year = {2014},
    date = {2014},
    publisher = {Apress},
    doi = {10.1007/978-1-4842-0076-6},
    url = {http://dx.doi.org/10.1007/978-1-4842-0076-6},
    note = {License: \href{https://creativecommons.org/licenses/by-nc/4.0/}{CC BY-NC}. Source:
}

@software{allaire2022,
    title = {Quarto},
    author = {Allaire, J.J. and Teague, Charles and Xie, Yihui and Dervieux, Christophe},
    year = {2022},
    month = {01},
    date = {2022-01-10},
    doi = {10.5281/zenodo.5960048},
    url = {https://github.com/quarto-dev/quarto-cli},
    version = {1.2},
    note = {License: \href{https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html}{GNU GPL
}

@article{millman2018,
    title = {Teaching Computational Reproducibility for Neuroimaging},
    author = {Millman, K. Jarrod and Brett, Matthew and Barnowski, Ross and Poline, Jean-Bapti
    year = {2018},
    month = {10},
    date = {2018-10-22},
    journal = {Frontiers in Neuroscience},
    volume = {12},
    doi = {10.3389/fnins.2018.00727},
    url = {https://dx.doi.org/10.3389/fnins.2018.00727},
    note = {License: \href{https://creativecommons.org/licenses/by/4.0/}{CC BY 4.0}. Website:
}

@article{milligan2014,
    title = {Introduction to the Bash Command Line},
    author = {Milligan, Ian and Baker, James},
    editor = {Crymble, Adam},
    year = {2014},
    month = {09},
```

**Listing 12.8** `quarto.yml`

```yaml
project:
  # project options (https://quarto.org/docs/reference/projects/options.html):
  type: book
  output-dir: _book
  execute-dir: project
```

**Listing 12.10** `variables.yml (excerpt)`

```yaml
links:
  email: mailto:lennart.wittkuhn@uni-hamburg.de
```

**Listing 12.11** `variables.yml`

```yaml
# https://quarto.org/docs/authoring/variables.html
links:
  email: mailto:lennart.wittkuhn@uni-hamburg.de
  github: https://github.com/lnnrtwttkhn/version-control-book
  issues: https://github.com/lnnrtwttkhn/version-control-book/issues
  site: https://lennartwittkuhn.com/version-control-book
  seminar: https://lennartwittkuhn.com/version-control-course-uhh-ws23/
  proposal: https://lennartwittkuhn.com/ddlitlab-proposal
  nextcloud: https://cloud.uni-hamburg.de/s/aD7NTNB9f4NDorT
  new_issues: https://github.com/lnnrtwttkhn/version-control-book/issues/new
  git: https://git-scm.com/
  markdown: https://www.markdownguide.org/
  quarto-workshop: https://lennartwittkuhn.com/quarto-workshop/
quiz:
  intro: https://version-control-versioncontrol.formr.org
  cli: https://version-control-cli.formr.org
  installation: https://version-control-installation.formr.org
  basics: https://version-control-basics.formr.org
  branches: https://version-control-branches.formr.org
  github: https://version-control-github.formr.org
uhh-ws23:
  intro: https://lennartwittkuhn.com/version-control-course-uhh-ws23/sessions/session01.html
  cli: https://lennartwittkuhn.com/version-control-course-uhh-ws23/sessions/session02.html
  basics: https://lennartwittkuhn.com/version-control-course-uhh-ws23/sessions/session03.html
  branches: https://lennartwittkuhn.com/version-control-course-uhh-ws23/sessions/session08.htm
  github: https://lennartwittkuhn.com/version-control-course-uhh-ws23/sessions/session09.html
  issues: https://lennartwittkuhn.com/version-control-course-uhh-ws23/sessions/session12.html
years: "2023 -- 2024"
language:
  dont:
    yes: "don't"
    no: "do not"
  folder:
    yes: "folder"
    no: "directory"
  documentation:
    yes: "documentation"
    no: "docs"
```