

# Introduction to version control with Git - Why we want to track versions and how to go back in time to a working version

This is the introductory lesson to version control using [Git](#). It is assumed to be the very first thing done in a course.

Our philosophy is that we start from own local repository, branching and merging (locally), and a brief introduction to pushing to remotes. In the separate [collaborative Git lesson](#), we teach more use of remote repositories and good collaborative workflows. We try to stick to simple workflows, just enough for researchers who are not obsessed with Git to be able to work well. We try to avoid commands which might get you into a confusing state.

The goals of the module as a whole are that the user will feel comfortable about staging changes, committing them, merging, and branching.

## Prerequisites

- A reasonably recent version of Git (ideal is 2.28 or newer but we recommend at least 2.23) is installed **and configured** ([installation instructions](#)). But also on older Git (2.0) the workshop will work and we will offer workarounds for Git below 2.28 or 2.23.
- For one of the episodes we need a [GitHub](#) user account (but alternatives exist, see below).
- Being comfortable with the command line. No expertise is required, but the lesson will be mostly taken from the command line. For most commands, where reasonable, we also offer the possibility to participate through the browser.
- To edit files on the local computer, learners should be familiar with using a **text editor** on their system. If you are new to text editors, we recommend to start with Nano or VS Code.

## Motivation

### Objectives

- Make sure nobody leaves the workshop without starting to use some form of version control.
- Discuss the reasons why we advocate distributed version control.

## Instructor note

- 15 min teaching/demonstration

## Git is all about keeping track of changes

We will learn how to keep track of changes first in a terminal ([example repository](#)):

```
commit 6eaba663debd9790a0f5b8a3ccb9deddd3b556cb
Author: Radovan Bast <bast@users.noreply.github.com>
Date: Thu Jun 22 17:28:19 2023 +0200

    add copyright and licensing information to each source file

commit c06ad3525060ab249bae44e6d83848b29779eb77
Author: Radovan Bast <bast@users.noreply.github.com>
Date: Thu Oct 21 20:16:33 2021 +0200

    collect imports

commit 98a2894d9f59ae045d117c1bdcc72a6948b3aa49
Author: Radovan Bast <bast@users.noreply.github.com>
Date: Sun Jun 14 16:56:16 2020 +0200

    autoformat with black

commit 3dc0c20f4ebdb519fa11b78aa18392e52e2fbb32
Author: Radovan Bast <bast@users.noreply.github.com>
Date: Sun Jun 14 16:30:57 2020 +0200

    fix for accepted_errors

    previously the code returned 1 before even getting
    to checking accepted/expected errors

commit f5d28243b7daaeff27a06202dcfc3b9b73c9f66c
Author: Radovan Bast <bast@users.noreply.github.com>
Date: Mon Sep 24 19:47:56 2018 +0200

    make it possible to launch binary using --launch-agent

commit 8a8c0563b2f52fb05ad03d806897a740767cf126
Author: Radovan Bast <bast@users.noreply.github.com>
Date: Mon Sep 24 19:38:28 2018 +0200




    fix for: 'str' object has no attribute 'decode'
```

Later also via web interface ([example repository](#)):




## Commits

History for [runtest](#) / [runtest](#) / [run.py](#)




Commits on Jun 22, 2023




**add copyright and licensing information to each source file**  
bast committed on Jun 22 ✓ 6eaba66   

Commits on Oct 21, 2021




**collect imports**  
bast committed on Oct 21, 2021 c06ad35   




Commits on Jun 14, 2020

**autoformat with black**  
bast committed on Jun 14, 2020 98a2894   

**fix for accepted\_errors** ...  
bast committed on Jun 14, 2020 3dc0c20   

Commits on Sep 24, 2018

**make it possible to launch binary using --launch-agent**  
bast committed on Sep 24, 2018 f5d2824   

**fix for: 'str' object has no attribute 'decode'**  
bast committed on Sep 24, 2018 8a8c056   

## Why do we need to keep track of versions?

Version control is an answer to these questions (do you recognize some of them?):

- “It broke ... hopefully I have a working version somewhere?”
- “Can you please send me the latest version?”
- “Where is the latest version?”
- “Which version are you using?”
- “Which version have the authors used in the paper I am trying to reproduce?”
- “Found a bug! Since when was it there?”
- “I am sure it used to work. When did it change?”
- “My laptop is gone. Is my thesis now gone?”

## Features: roll-back, branching, merging, collaboration

- **Roll-back:** you can always go back to a previous version and compare
- **Branching and merging:**
  - Work on different ideas at the same time
  - Different people can work on the same code/project without interfering
  - You can experiment with an idea and discard it if it turns out to be a bad idea

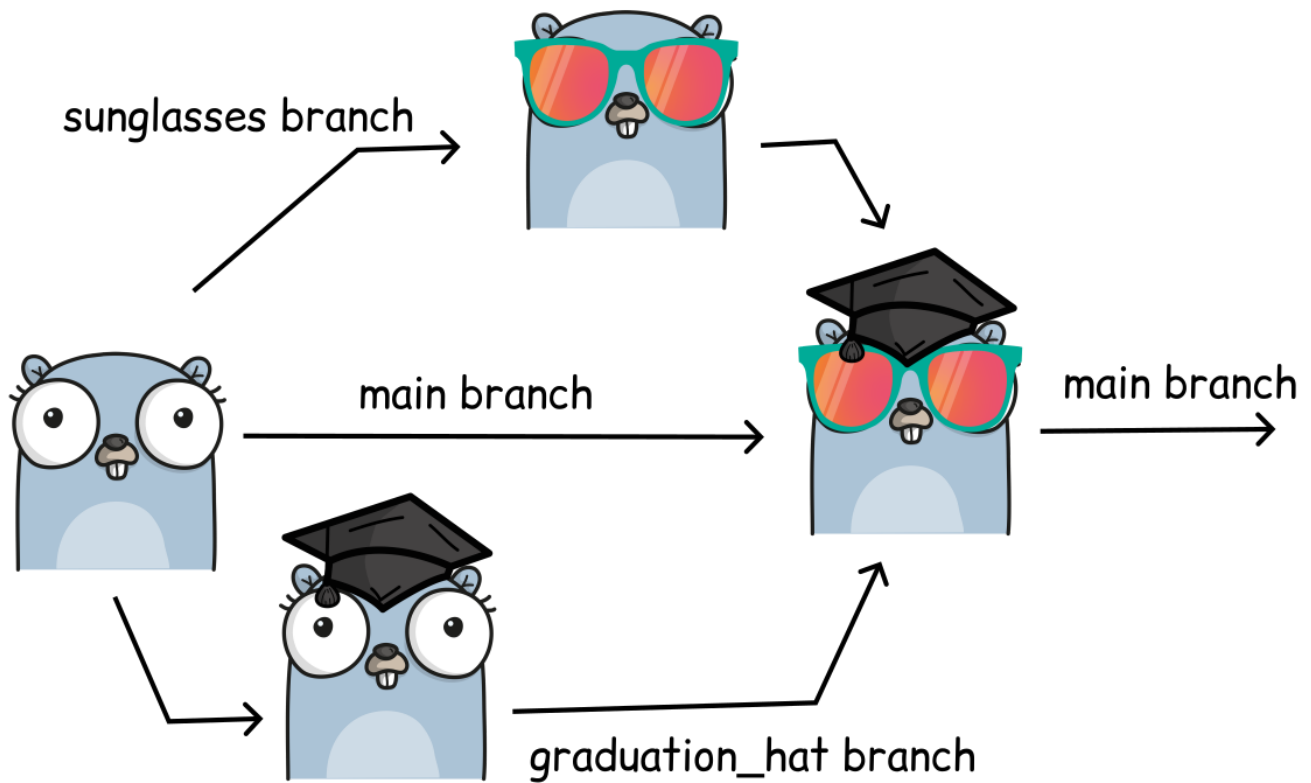


Image created using <https://gopherize.me/> (inspiration).

- **Collaboration:** review, compare, share, discuss
- [Example network graph](#)

## Reproducibility

- How do you indicate which version of your code you have used in your paper?
- When you find a bug, how do you know **when precisely** this bug was introduced (Are published results affected? Do you need to inform collaborators or users of your code?).

With version control we can “annotate” code ([browse this example online](#)):

networkx / networkx / algorithms / boundary.py

Ignoring revisions in .git-blame-ignore-revs.

eriknw Add @nx.\_dispatch decorator to most algorithms (#6688) fae8af6 · 2 weeks ago History

Code Blame 167 lines (129 loc) · 5.21 KB

Older Newer Contributors 12

8 years ago	Adds functions for measuring ...	1	"""Routines to find the boundary of a set of nodes.
		2	
		3	An edge boundary is a set of edges, each of which has exactly one
		4	endpoint in a given set of nodes (or, in the case of directed graphs,
		5	the set of edges whose source node is in the set).
15 years ago	Merged revisions 741-766,769-770...	6	
8 years ago	Adds functions for measuring ...	7	A node boundary of a set *S* of nodes is the set of (out-)neighbors of
		8	nodes in *S* that are outside *S*.
15 years ago	Merged revisions 741-766,769-770...	9	
8 years ago	Adds functions for measuring ...	10	"""
		11	from itertools import chain
15 years ago	Merged revisions 741-766,769-770...	12	
9 months ago	plugin based backend infrastr...	13	import networkx as nx
		14	
8 years ago	Adds functions for measuring ...	15	__all__ = ["edge_boundary", "node_boundary"]
		16	
		17	
2 weeks ago	Add @nx._dispatch decorato...	18	@nx._dispatch(edge_attrs={"data": "default"}, preserve_edge_attrs="data")
8 years ago	Adds functions for measuring ...	19	def edge_boundary(G, nbunch1, nbunch2=None, data=False, keys=False, default=None):
8 years ago	Change default role for sphinx...	20	"""Returns the edge boundary of `nbunch1`.
8 years ago	Adds functions for measuring ...	21	
		22	The *edge boundary* of a set *S* with respect to a set *T* is the
		23	set of edges (*u*, *v*) such that *u* is in *S* and *v* is in *T*.
		24	If *T* is not specified, it is assumed to be the set of all nodes
		25	not in *S*.
15 years ago	update rst and doc strings for ...	26	
		27	Parameters
8 years ago	Minor docstring fixes.	28	-----
8 years ago	Adds functions for measuring ...	29	G : NetworkX graph
15 years ago	update rst and doc strings for ...	30	

Example of a git-annotated code with code and history side-by-side.

## Talking about code

Which of these two is more practical?

- “Clone the code, go to the file ‘src/util.rs’, and search for ‘time\_iso8601’”. Oh! But make sure you use the version from August 2023.”
- Or I can send you a [permlink](#):

```

37  #[cfg(test)]
38  pub(crate) use set;
39
40  // Get current time as an ISO time stamp.
41  pub fn time_iso8601() -> String {
42      let local_time = Local::now();
43      format!("{}", local_time.format("%Y-%m-%dT%H:%M:%SZ"))
44  }
45
46  // Carve up a line of text into space-separated chunks + the start indices of the chunks.
47  pub fn chunks(input: &str) -> (Vec<usize>, Vec<&str>) {
48      let mut start_indices: Vec<usize> = Vec::new();

```

Permalink that points to a code portion.

## What we typically like to snapshot

- Software (this is how it started but Git/GitHub can track a lot more)
- Scripts
- Documents (plain text files much better suitable than Word documents)
- Manuscripts (Git is great for collaborating/sharing LaTeX or [Quarto](#) manuscripts)
- Configuration files
- Website sources
- Data

### Discussion

In this example somebody tried to keep track of versions without a version control system tool like Git. Discuss the following directory listing. What possible problems do you anticipate with this kind of “version control”:

```
myproject-2019.zip
myproject-2020-February.zip
myproject-2021-August.zip
myproject-2023-09-19-working.zip
myproject-2023-09-21.zip
myproject-2023-09-21-test.zip
myproject-2023-09-21-myversion.zip
myproject-2023-09-21-newfeature.zip
...
```

### ✓ Solution

- Giving a version to a collaborator and merging changes later with own changes sounds like lots of work.
- What if you discover a bug and want to know since when the bug existed?

## Difficulties of version control

Despite the benefits, let's be honest, there are some difficulties:

- One more thing to learn (it's probably worth it and will save you more time in the long run; basic career skill).
- Difficult if your collaborators don't want to use it (in the worst case, you can version control on your side and email them versions).
- Advanced things can be difficult, but basics are often enough (ask others for help when needed).

### Why Git and not another tool?

- **Easy to set up:** no server needed.
- **Very popular:** chances are high you will need to contribute to somebody else's code which is tracked with Git.
- **Distributed:** good backup, no single point of failure, you can track and clean-up changes offline, simplifies collaboration model for open-source projects.
- Important **platforms** such as [GitHub](#), [GitLab](#), and [Bitbucket](#) build on top of Git.

However, any version control is better than no version control and it is OK to prefer a different tool than Git such as [Subversion](#), [Mercurial](#), [Pijul](#), or others.

## Basics

### ! Objectives

- Learn to create Git repositories and make commits.
- Get a grasp of the structure of a repository.
- Learn how to inspect the project history.
- Learn how to write useful commit log messages.

### Instructor note

- 35 min teaching/type-along
- 40 min exercise

## What is Git, and what is a Git repository?

- Git is a version control system: can **record/save snapshots** and track the content of a folder as it changes over time.
- Every time we **commit** a snapshot, Git records a snapshot of the **entire project**, saves it, and assigns it a version.
- These snapshots are kept inside a sub-folder called `.git`.
- If we remove `.git`, we remove the repository and history (but keep the working directory!).
- The directory `.git` uses relative paths - you can move the whole repository somewhere else and it will still work.
- Git doesn't do anything unless you ask it to (it **does not record anything automatically**).
- Multiple interfaces to Git exist (command line, graphical interfaces, web interfaces).

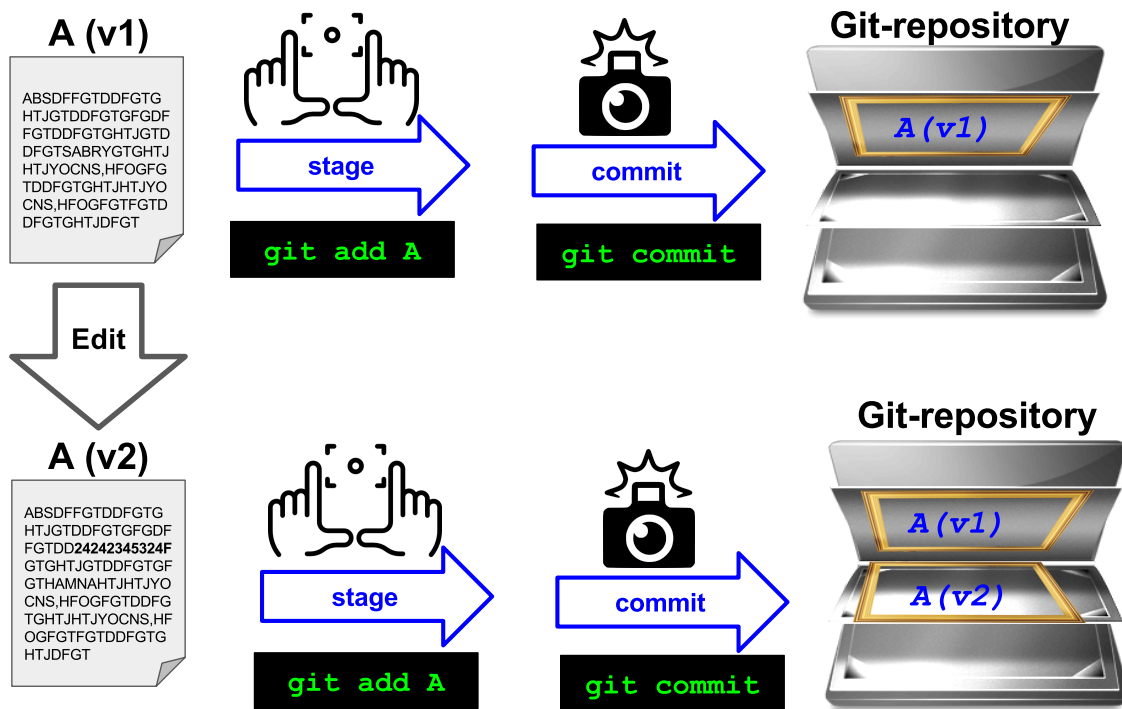
## Recording a snapshot with Git

- Git takes snapshots only if we request it.
- We will record changes in two steps (we will later explain why this is a recommended practice).
- Example (we don't need to type yet):

```
$ git add FILE.txt
$ git commit

$ git add FILE.txt ANOTHERFILE.txt
$ git commit
```

- We first focus (`git add`), we “stage” the change, then record (`git commit`):



*Git staging and committing.*

### Question for the more advanced participants

What do you think will be the outcome if you stage a file and then edit it and stage it again, do this several times and at the end perform a commit? Think of focusing several scenes and pressing the shutter at the end.

## Configuring Git command line

Before we start using Git on the command line, we need to configure Git. This is also part of the [installation instructions](#) but we need to make sure we all have set name, email address, editor, and default branch:

```
$ git config --global user.name "Your Name"
$ git config --global user.email yourname@example.com
$ git config --global core.editor nano
$ git config --global init.defaultBranch main
```

Verify with:



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

### Instructor note

Instructors, give learners enough time to do the above configuration steps.

## Type-along: Tracking a guacamole recipe with Git

We will learn how to initialize a Git repository, how to track changes, and how to make delicious guacamole! (Inspiration for this example based on a suggestion by B. Smith in a discussion in the Carpentries mailing list)

The motivation for taking a cooking recipe instead of a program is that everybody can relate to cooking but not everybody may be able to relate to a program written in e.g. Python or another specific language.

### Instructor note

Instructors, please encourage now that participants type along.

### Note

It is possible to go through this lesson in the command line or in the browser (on GitHub).

- We recommend to start with the command line but later to also try in the browser.
- If you get really stuck in the command line, try following in the browser and later you can try to return to the command line.

## Creating a repository

One of the basic principles of Git is that it is **easy to create repositories**:

### Command line

### Browser (GitHub)

```
$ mkdir recipe
$ cd recipe
$ git init -b main
```

That's it! With `git init -b main` have now created an empty Git repository where `main` is the default branch (more about branches later).

We will use `git status` a lot to check out what is going on:

```
$ git status

On branch main

No commits yet

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

We will make sense of this information during this workshop.

## Adding files and committing changes

Let us now create two files.

One file is called `ingredients.txt` and contains:

```
* 2 avocados
* 1 chili
* 1 lime
* 2 tsp salt
```

The second file is called `instructions.txt` and contains:

```
* chop avocados
* chop onion
* chop chili
* squeeze lime
* add salt
* and mix well
```

Command line

Browser (GitHub)

As mentioned above, in Git you can always check the status of files in your repository using `git status`. It is always a safe command to run and in general a good idea to do when you are trying to figure out what to do next:

```
$ git status

On branch main

No commits yet

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

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

The two files are untracked in the repository (directory). You want to **add the files** (focus the camera) to the list of files tracked by Git. Git does not track any files automatically and you need make a conscious decision to add a file. Let's do what Git hints at, and add the files, one by one:

```
$ git add ingredients.txt
$ git status

On branch main

No commits yet

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

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

Now this change is *staged* and ready to be committed. Let us now commit the change to the repository:

```
$ git commit -m "adding ingredients"

[main (root-commit) f146d25] adding ingredients
1 file changed, 4 insertions(+)
create mode 100644 ingredients.txt
```

Right after we query the status to get this useful command into our muscle memory:

```
$ git status
```

Now stage and commit also the other file:

```
$ git add instructions.txt
$ git commit -m "adding instructions"
```

We will add a third file to the repository, `README.md`, containing:

```
# recipe

This is an exercise repository.
```

Now stage and commit also the `README.md` file:

```
$ git add README.md
$ git commit -m "adding README"
```

What does the `-m` flag mean? Let us check the help page for that command:

```
$ git help commit
```

You should see a very long help page as the tool is very versatile (press q to quit). Do not worry about this now but keep in mind that you can always read the help files when in doubt. Searching online can also be useful, but choosing search terms to find relevant information takes some practice and discussions in some online threads may be confusing. Note that help pages also work when you don't have a network connection!

## Exercise: Record changes



### Basic-1: Record changes

Add 1/2 onion to `ingredients.txt` and also the instruction to "enjoy!" to `instructions.txt`.

Command line

Browser (GitHub)

After modifying the files, do not stage the changes yet (do not `git add` yet).

When you are done editing the files, try `git diff`:

```
$ git diff
```

You will see (can you identify in there the two added lines?):

```
diff --git a/ingredients.txt b/ingredients.txt
index 4422a31..ba8854f 100644
--- a/ingredients.txt
+++ b/ingredients.txt
@@ -2,3 +2,4 @@
 * 1 chili
 * 1 lime
 * 2 tsp salt
+* 1/2 onion
diff --git a/instructions.txt b/instructions.txt
index 7811273..2b11074 100644
--- a/instructions.txt
+++ b/instructions.txt
@@ -4,3 +4,4 @@
 * squeeze lime
 * add salt
 * and mix well
+* enjoy!
```

Now first stage and commit each change separately (what happens when we leave out the `-m` flag?):

```
$ git add ingredients.txt
$ git commit -m "add half an onion"
$ git add instructions.txt
$ git commit # <-- we have left out -m "..."
```

When you leave out the `-m` flag, Git should open an editor where you can edit your commit message. This message will be associated and stored with the changes you made. This message is your chance to explain what you've done and convince others (and your future self) that the changes you made were justified. Write a message and save and close the file.

When you are done committing the changes, experiment with these commands:

```
$ git log
$ git log --stat
$ git log --oneline
```

# Git history and log

Command line

Browser (GitHub)

If you haven't yet, please try now `git log`:

```
$ git log
```

```
commit e7cf023efe382340e5284c278c6ae2c087dd3ff7 (HEAD -> main)
Author: Radovan Bast <bast@users.noreply.github.com>
Date:   Sun Sep 17 19:12:47 2023 +0200
```

```
    don't forget to enjoy
```

```
commit 79161b6e67c62ad4688a58c1e54183334611a390
Author: Radovan Bast <bast@users.noreply.github.com>
Date:   Sun Sep 17 19:12:32 2023 +0200
```

```
    add half an onion
```

```
commit a3394e39535343c4dae3bb4f703741a31aa8b78a
Author: Radovan Bast <bast@users.noreply.github.com>
Date:   Sun Sep 17 18:47:14 2023 +0200
```

```
    adding README
```

```
commit 369624674e63de48055a65bf63055bd59c985d22
Author: Radovan Bast <bast@users.noreply.github.com>
Date:   Sun Sep 17 18:46:58 2023 +0200
```

```
    adding instructions
```

```
commit f146d25b94569a15e94d7f0da6f15d7554f76c49
Author: Radovan Bast <bast@users.noreply.github.com>
Date:   Sun Sep 17 18:35:52 2023 +0200
```

```
    adding ingredients
```

- We can browse the development and access each state that we have committed.
- The long hashes uniquely label a state of the code.
- They are not just integers counting 1, 2, 3, 4, ... (why?).
- Output is in reverse chronological order, i.e. **newest commits on top**.
- We will use them when comparing versions and when going back in time.
- `git log --oneline` only shows the first 7 characters of the commit hash and is good to get an overview.
- If the first characters of the hash are unique it is not necessary to type the entire hash.
- `git log --stat` is nice to show which files have been modified.

## Optional exercises: Comparing changes

### 🔧 (optional) Basic-2: Comparing and showing commits

#### Command line

#### Browser (GitHub)

1. Have a look at specific commits with `git show HASH`.
2. Inspect differences between commit hashes with `git diff HASH1 HASH2`.

### 🔧 (optional) Basic-3: Visual diff tools

This exercise is only relevant for the command line. In the browser, the preview is already side-by-side and “visual”.

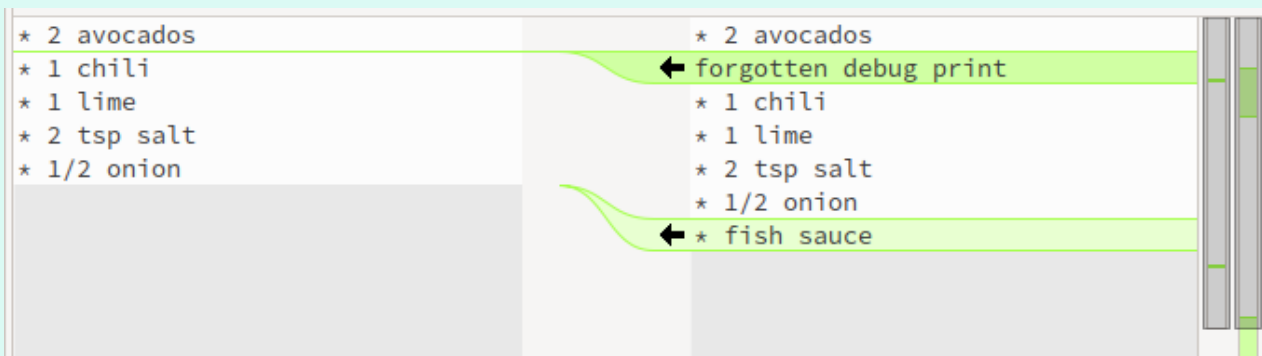
- Make further modifications and experiment with `git difftool` (requires installing one of the [visual diff tools](#)):

On Windows or Linux:

```
$ git difftool --tool=meld HASH
```

On macOS:

```
$ git difftool --tool=opendiff HASH
```



*Git difftool using meld.*

You probably want to use the same visual diff tool every time and you can configure Git for that:

```
$ git config --global diff.tool meld
```

#### (optional) Basic-4: Browser and command line

You have noticed that it is possible to work either in the command line or in the browser. It could help to deepen the understanding trying to do the above steps in both.

- If you have managed to do the above in the command line, try now in the browser.
- If you got stuck in the command line and move to the browser, try now to troubleshoot the command line Git.

## Writing useful commit messages

Using `git log --oneline` or browsing a repository on the web, we better understand that the first line of the commit message is very important.

Good example:

```
increase threshold alpha to 2.0

the motivation for this change is
to enable ...
...
this is based on a discussion in #123
```

Convention: **one line summarizing the commit, then one empty line, then paragraph(s) with more details in free form, if necessary.**

- **Why something was changed is more important than what has changed.**
- Cross-reference to issues and discussions if possible/relevant.
- Bad commit messages: “fix”, “oops”, “save work”
- Bad examples: <http://whatthecommit.com>
- Write commit messages in English that will be understood 15 years from now by someone else than you. Or by your future you.
- Many projects start out as projects “just for me” and end up to be successful projects that are developed by 50 people over decades.
- [Commits with multiple authors](#) are possible.

Good references:

- [“My favourite Git commit”](#)
- [“On commit messages”](#)
- [“How to Write a Git Commit Message”](#)



## Note

A great way to learn how to write commit messages and to get inspired by their style choices: **browse repositories of codes that you use/like:**

Some examples (but there are so many good examples):

- [SciPy](#)
- [NumPy](#)
- [Pandas](#)
- [Julia](#)
- [ggplot2](#), compare with their [release notes](#)
- [Flask](#), compare with their [release notes](#)

When designing commit message styles consider also these:

- How will you easily generate a changelog or release notes?
- During code review, you can help each other improving commit messages.

But remember: it is better to make any commit, than no commit. Especially in small projects. **Let not the perfect be the enemy of the good enough.**

## Ignoring files and paths with `.gitignore`

### Discussion

- Should we add and track all files in a project?
- How about generated files?
- Why is it considered a bad idea to commit compiled binaries to version control?
- What types of generated files do you know?

Compiled and generated files are not committed to version control. There are many reasons for this:

- These files can make it more difficult to run on different platforms.
- These files are automatically generated and thus do not contribute in any meaningful way.
- When tracking generated files you could see differences in the code although you haven't touched the code.

For this we use `.gitignore` files. Example:

```
# ignore compiled python 2 files
*.pyc
# ignore compiled python 3 files
__pycache__
```

An example taken from the [official Git documentation](#):

```
# ignore objects and archives, anywhere in the tree.  
*.[oa]  
# ignore generated html files,  
*.html  
# except foo.html which is maintained by hand  
!foo.html  
# ignore everything under build directory  
build/
```

- `.gitignore` should be part of the repository because we want to make sure that all developers see the same behavior.
- **All files should be either tracked or ignored.**
- `.gitignore` uses something called a [shell glob syntax](#) for determining file patterns to ignore. You can read more about the syntax in the [documentation](#).
- You can have `.gitignore` files in lower level directories and they affect the paths below.

## Graphical user interfaces

We have seen how to make commits in the command line and via the GitHub website. But it is also possible to work from within a Git graphical user interface (GUI):

- [GitHub Desktop](#)
- [SourceTree](#)
- [List of third-party GUIs](#)

## Summary

Now we know how to save snapshots:

```
$ git add FILE(S)  
$ git commit
```

And this is what we do as we program.

Every state is then saved and later we will learn how to go back to these “checkpoints” and how to undo things.

```
$ git init -b main # initialize new repository (main is default branch)
$ git add          # add files or stage file(s)
$ git commit       # commit staged file(s)
$ git status       # see what is going on
$ git log          # see history
$ git diff         # show unstaged/uncommitted modifications
$ git show         # show the change for a specific commit
$ git mv           # move/rename tracked files
$ git rm           # remove tracked files
```

Git is not ideal for large binary files (for this consider [git-annex](#)).

### Basic-5: Test your understanding

Which command(s) below would save the changes of `myfile.txt` to an existing local Git repository?

1. `$ git commit -m "my recent changes"`
2. `$ git init myfile.txt`  
`$ git commit -m "my recent changes"`
3. `$ git add myfile.txt`  
`$ git commit -m "my recent changes"`
4. `$ git commit -m myfile.txt "my recent changes"`

### ✓ Solution

1. Would only create a commit if files have already been staged.
2. Would try to create a new repository in a folder "myfile.txt".
3. **Is correct: first add the file to the staging area, then commit.**
4. Would try to commit a file "my recent changes" with the message myfile.txt.

### Keypoints

- It takes only one command to initialize a Git repository: `git init -b main`.
- Commits should be used to tell a story.
- Git uses the `.git` folder to store the snapshots.

- Don't be afraid to stage and commit often. Better too often than not often enough.

## Branching and merging

### Objectives

- Be able to create and merge branches.
- Know the difference between a branch and a tag.

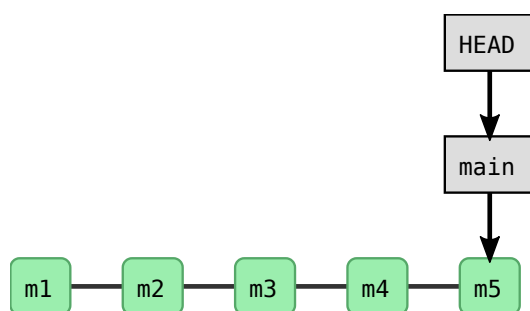
### Instructor note

- 30 min teaching/type-along
- 20 min exercise

## Motivation for branches

In the previous section we tracked a guacamole recipe with Git.

Up until now our repository had only one branch with one commit coming after the other:



*Linear Git repository.*

- Commits are depicted here as little boxes with abbreviated hashes.
- Here the branch `main` points to a commit.
- “HEAD” is the current position (remember the recording head of tape recorders?). When we say `HEAD`, we mean those literal letters - this isn't a placeholder for something else.
- When we talk about branches, we often mean all parent commits, not only the commit pointed to.

Now we want to do this:

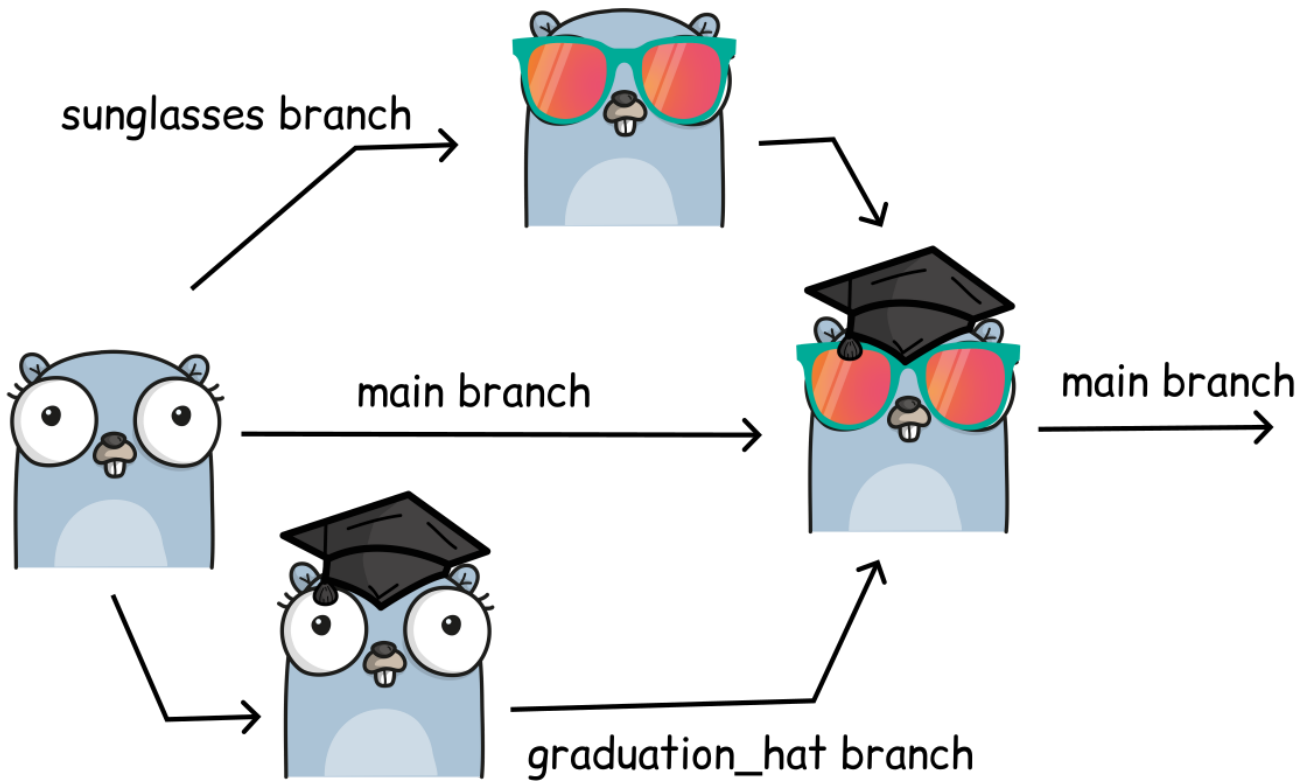
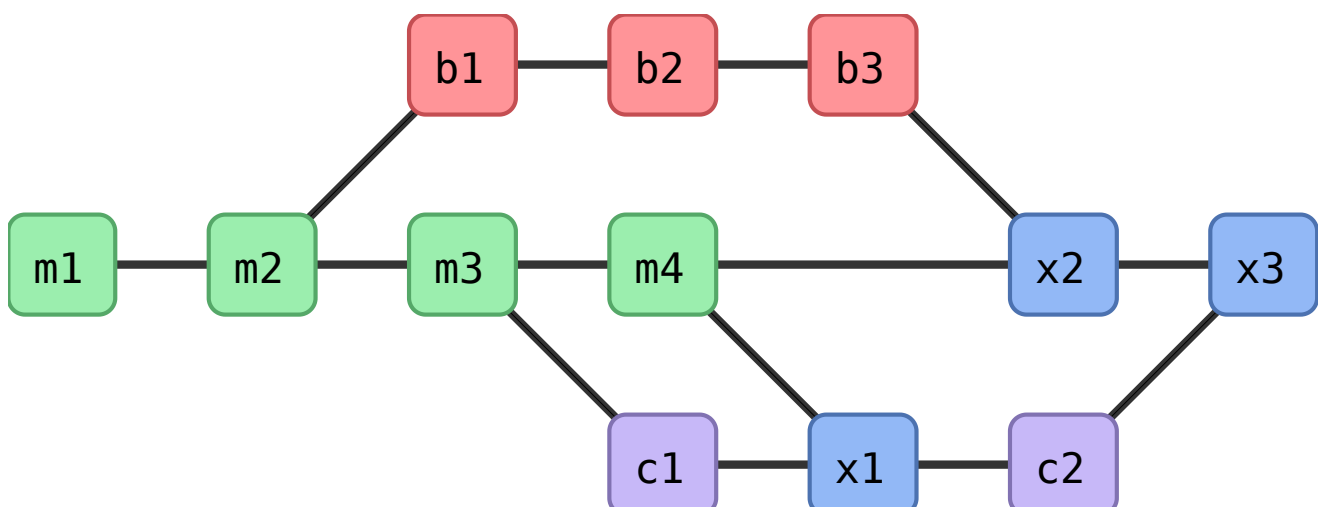


Image created using <https://gopherize.me/> (inspiration).

Software development is often not linear:

- We typically need at least one version of the code to “work” (to compile, to give expected results, ...).
- At the same time we work on new features, often several features concurrently. Often they are unfinished.
- We need to be able to separate different lines of work really well.

The strength of version control is that it permits the researcher to **isolate different tracks of work**, which can later be merged to create a composite version that contains all changes:



Isolated tracks of work.

- We see branching points and merging points.
- Main line development is often called `main` or `master`.
- Other than this convention there is nothing special about `main` or `master`, it is a branch like any other.
- Commits form a directed acyclic graph (we have left out the arrows to avoid confusion about the time arrow).

A group of commits that create a single narrative are called a **branch**. There are different branching strategies, but it is useful to think that a branch tells the story of a feature, e.g. “fast sequence extraction” or “Python interface” or “fixing bug in matrix inversion algorithm”.

### ! An important alias

We will now define an *alias* in Git, to be able to nicely visualize branch structure in the terminal without having to remember a long Git command (more details about aliases are given in a later section). **This is extensively used in the rest of this and other lessons:**

```
$ git config --global alias.graph "log --all --graph --decorate --oneline"
```

### Instructor note

Instructors, please demonstrate how to set this alias and ensure that all create it. This is very important for this lesson and git-collaborative.

Let us inspect the project history using the `git graph` alias:

```
$ git graph
* e7cf023 (HEAD -> main) don't forget to enjoy
* 79161b6 add half an onion
* a3394e3 adding README
* 3696246 adding instructions
* f146d25 adding ingredients
```

- We have a couple commits and only one development line (branch) and this branch is called `main`.
- Commits are states characterized by a 40-character hash (checksum).
- `git graph` print abbreviations of these checksums.
- **Branches are pointers that point to a commit.**
- Branch `main` points to a commit (in this example it is `e7cf023efe382340e5284c278c6ae2c087dd3fff7` but on your computer the hash will be different).

- `HEAD` is another pointer, it points to where we are right now (currently `main` )

In the following we will learn how to create branches, how to switch between them, how to merge branches, and how to remove them afterwards.

## Creating and working with branches

### Instructor note

We do the following part together. Encourage participants to type along.

### ❗ It is possible to create and merge branches directly on GitHub

- However, we do not have screenshots for that in this episode
- But if you prefer to work in the browser, please try it
- Please contribute screenshots to this lesson

Let's create a branch called `experiment` where we add cilantro to `ingredients.txt` (text after `"#"` are comments and not part of the command).

```
$ git branch experiment main  # creates branch "experiment" from "main"
$ git switch experiment      # switch to branch "experiment"
$ git branch                 # list all local branches and show on which branch we are
```

### ❗ Note

In case `git switch` does not work, your Git version might be older than from 2019. On older Git it is `git checkout` instead of `git switch` .

- Verify that you are on the `experiment` branch (note that `git graph` also makes it clear what branch you are on: `HEAD -> branchname` ):

```
$ git branch

* experiment
  main
```

This command shows where we are, it does not create a branch.

- Then add 2 tbsp cilantro **on top** of the `ingredients.txt` :

```
* 2 tbsp cilantro
* 2 avocados
* 1 chili
* 1 lime
* 2 tsp salt
* 1/2 onion
```

- Stage this and commit it with the message “let us try with some cilantro”.
- Then reduce the amount of cilantro to 1 tbsp, stage and commit again with “maybe little bit less cilantro”.

We have created **two new commits**:

```
$ git graph
```

```
* bcb8b78 (HEAD -> experiment) maybe little bit less cilantro
* f6ec7b7 let us try with some cilantro
* e7cf023 (main) don't forget to enjoy
* 79161b6 add half an onion
* a3394e3 adding README
* 3696246 adding instructions
* f146d25 adding ingredients
```

- The branch `experiment` is two commits ahead of `main`.
- We commit our changes to this branch.

## Exercise: Create and commit to branches

### Branch-1: Create and commit to branches

In this exercise, you will create another new branch and few more commits. We will use this in the next section, to practice merging. **The goal of the exercise is to end up with 3 branches.**

- Change to the branch `main`.
- Create another branch called `less-salt`.
  - Note! makes sure you are on main branch when you create the `less-salt` branch.
  - A safer way would be to explicitly mention to create from the main branch as shown below:

```
$ git branch less-salt main
```

- Switch to the `less-salt` branch.
- On the `less-salt` branch reduce the amount of salt.
- Commit your changes to the `less-salt` branch.



Use the same commands as we used above.

We now have three branches (in this case `HEAD` points to `less-salt`):

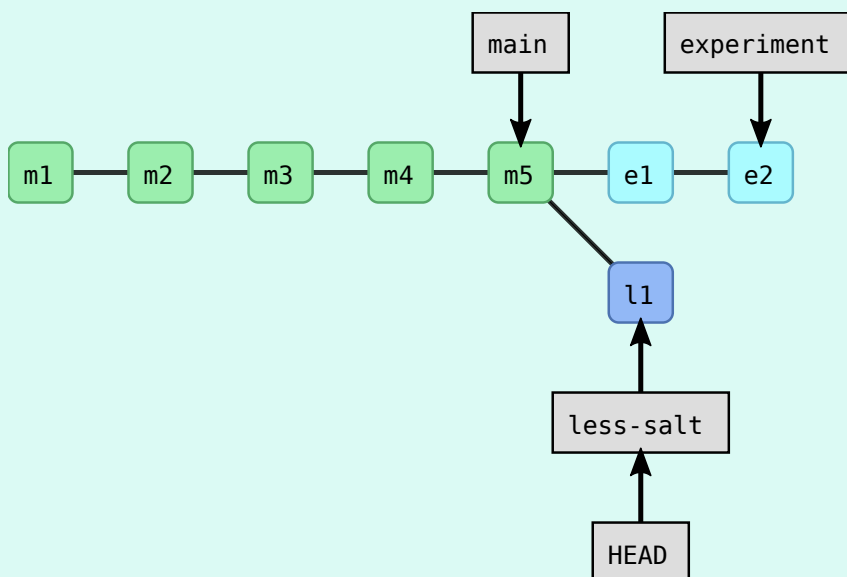
```
$ git branch

experiment
* less-salt
main

$ git graph

* bf28166 (HEAD -> less-salt) reduce amount of salt
| * bcb8b78 (experiment) maybe little bit less cilantro
| * f6ec7b7 let us try with some cilantro
|/
* e7cf023 (main) don't forget to enjoy
* 79161b6 add half an onion
* a3394e3 adding README
* 3696246 adding instructions
* f146d25 adding ingredients
```

Here is a graphical representation of what we have created:



- Now switch to `main`.
- In a new commit, improve the `README.md` file (we added the word “Guacamole”):

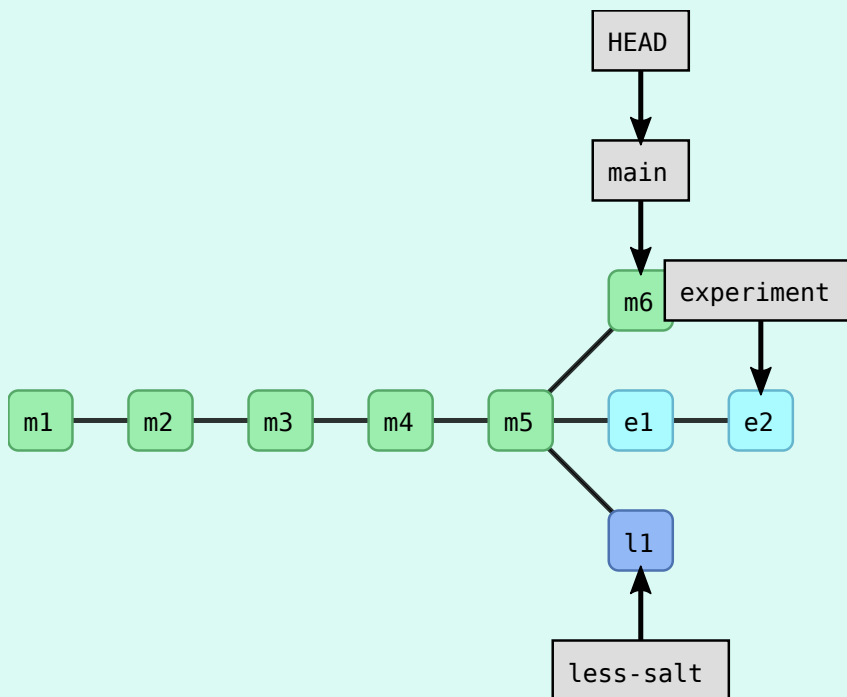
```
# Guacamole recipe

This is an exercise repository.
```

Now you should have this situation:

```
$ git graph
```

```
* b4af65b (HEAD -> main) improve the documentation
| * bf28166 (less-salt) reduce amount of salt
|/
| * bcb8b78 (experiment) maybe little bit less cilantro
| * f6ec7b7 let us try with some cilantro
|/
* e7cf023 don't forget to enjoy
* 79161b6 add half an onion
* a3394e3 adding README
* 3696246 adding instructions
* f146d25 adding ingredients
```



And for comparison this is how it looks [on GitHub](#).

## Exercise: Merging branches

It turned out that our experiment with cilantro was a good idea. Our goal now is to merge `experiment` into `main`.

### Branch-2: Merge branches

Merge `experiment` and `less-salt` back into `main` following the lesson below until the point where we start deleting branches.

**!** If you got stuck in the above exercises or joined later

If you got stuck in the above exercises or joined later, you can apply the commands below. But skip this box if you managed to create branches.

```

$ cd .. # step out of the current directory

$ git clone https://github.com/coderefinery/recipe-before-merge.git
$ cd recipe-before-merge

$ git switch experiment
$ git switch less-salt
$ git switch main

$ git remote remove origin

$ git graph

```

Or call a helper to un-stuck it for you.

First we make sure we are on the branch we wish to merge **into**:

```

$ git branch

  experiment
  less-salt
* main

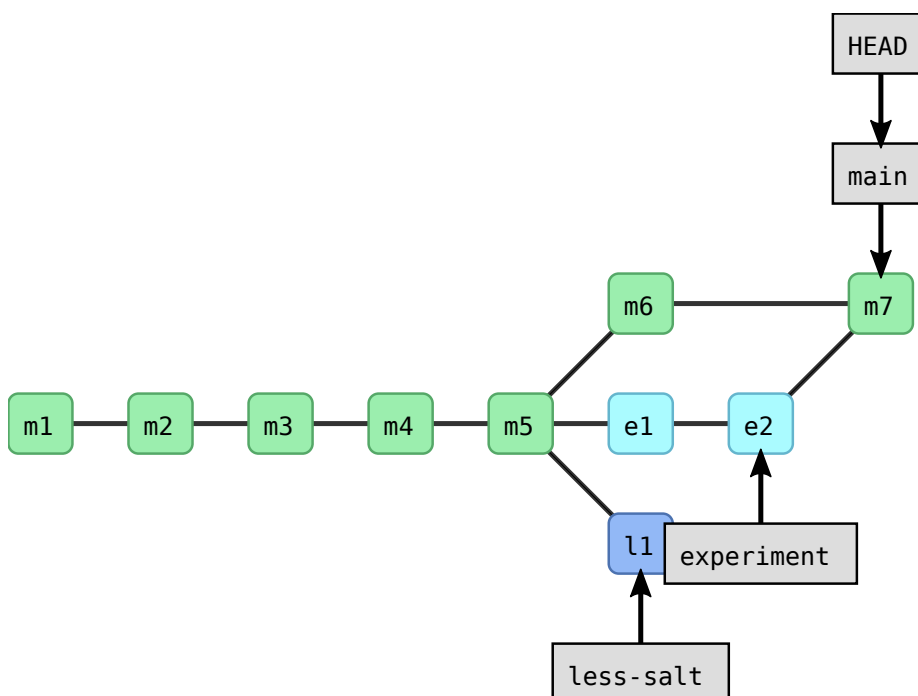
```

Then we merge `experiment` into `main` :

```

$ git merge experiment

```



We can verify the result:

```
$ git graph

* 81fcc0c (HEAD -> main) Merge branch 'experiment'
|\
| * bcb8b78 (experiment) maybe little bit less cilantro
| * f6ec7b7 let us try with some cilantro
* | b4af65b improve the documentation
|/
| * bf28166 (less-salt) reduce amount of salt
|/
* e7cf023 don't forget to enjoy
* 79161b6 add half an onion
* a3394e3 adding README
* 3696246 adding instructions
* f146d25 adding ingredients
```

What happens internally when you merge two branches is that Git creates a new commit, attempts to incorporate changes from both branches and records the state of all files in the new commit. While a regular commit has one parent, a merge commit has two (or more) parents.

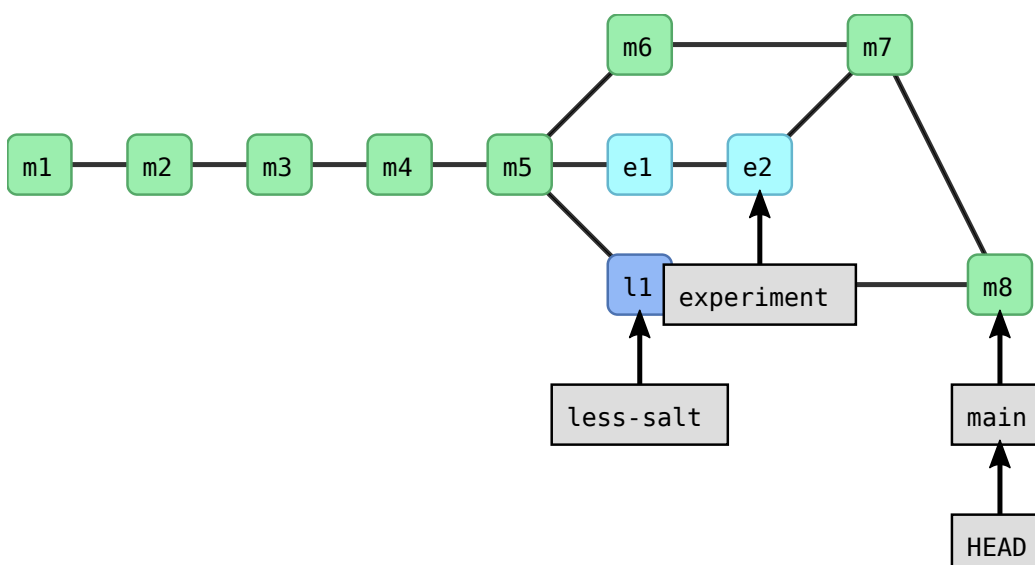
To view the branches that are merged into the current branch we can use the command:

```
$ git branch --merged

experiment
* main
```

We are also happy with the work on the `less-salt` branch. Let us merge that one, too, into `main`:

```
$ git branch # make sure you are on main
$ git merge less-salt
```



*Commit graph after merge.*

We can verify the result in the terminal:

```
$ git graph

*    4e03d4b (HEAD -> main) Merge branch 'less-salt'
| \
| * bf28166 (less-salt) reduce amount of salt
* |    81fcc0c Merge branch 'experiment'
| \ \
| * | bcb8b78 (experiment) maybe little bit less cilantro
| * | f6ec7b7 let us try with some cilantro
| | /
* / b4af65b improve the documentation
| /
* e7cf023 don't forget to enjoy
* 79161b6 add half an onion
* a3394e3 adding README
* 3696246 adding instructions
* f146d25 adding ingredients
```

Observe how Git nicely merged the changed amount of salt and the new ingredient **in the same file without us merging it manually**:

```
$ cat ingredients.txt

* 1 tbsp cilantro
* 2 avocados
* 1 chili
* 1 lime
* 1 tsp salt
* 1/2 onion
```

If the same file is changed in both branches, Git attempts to incorporate both changes into the merged file. If the changes overlap then the user has to manually *settle merge conflicts* (we will do that later).

## Deleting branches safely

Both feature branches are merged:

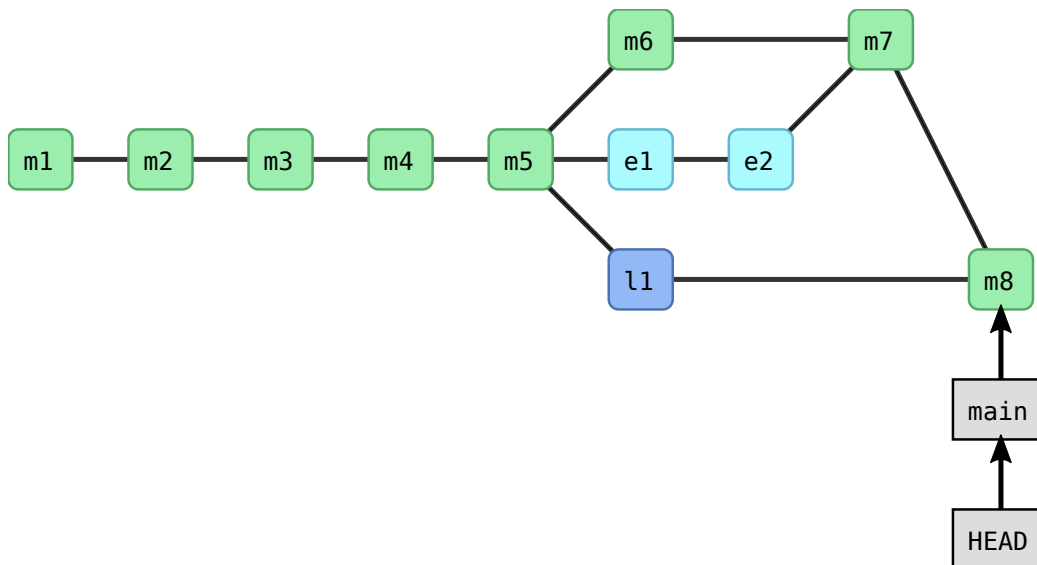
```
$ git branch --merged

experiment
less-salt
* main
```

This means we can delete the branches:

```
$ git branch -d experiment
$ git branch -d less-salt
```

This is the result:



*Commit graph after merged branches were deleted.*

We observe that when deleting branches, only the pointers (“sticky notes”) disappeared, not the commits.

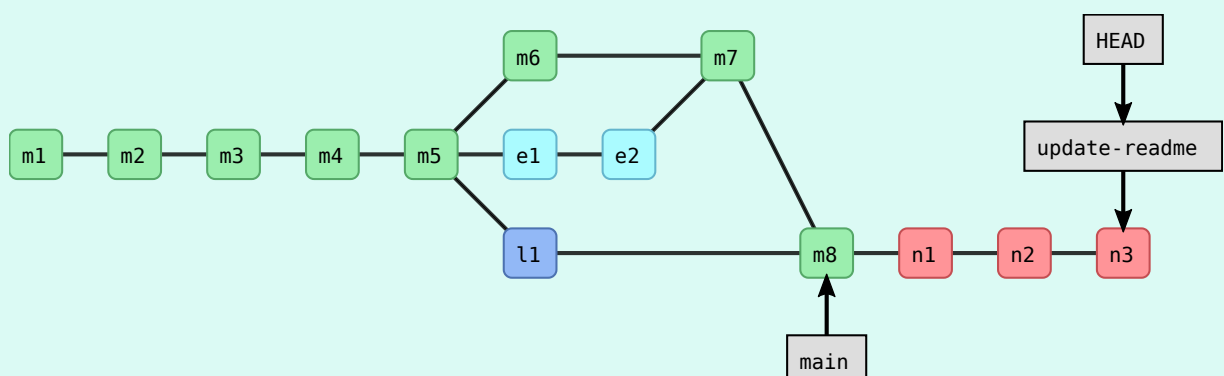
Git will not let you delete a branch which has not been reintegrated unless you insist using `git branch -D`. Even then your commits will not be lost but you may have a hard time finding them as there is no branch pointing to them.

## Optional exercises with branches

The following exercises are more advanced, absolutely no problem to postpone them to a few months later. If you give them a go, keep in mind that you might run into conflicts, which we will learn to resolve in the next section.

### (optional) Branch-3: Perform a fast-forward merge

1. Create a new branch from `main` and switch to it.
2. Create a couple of commits on the new branch (for instance edit `README.md`):



3. Now switch to `main`.
4. Merge the new branch to `main`.
5. Examine the result with `git graph`.
6. Have you expected the result? Discuss what you see.

### ✓ Solution

You will see that in this case no merge commit was created and Git merged the two branches by moving (fast-forwarding) the “main” branch (label) three commits forward.

This was possible since one branch is the ancestor of the other and their developments did not diverge.

A merge that does not require any merge commit is a fast-forward merge.

### 🔧 (optional) Branch-4: Rebasing a branch (instead of merge)

As an alternative to merging branches, one can also *rebase* branches. Rebasing means that the new commits are *replayed* on top of another branch (instead of creating an explicit merge commit). **Note that rebasing changes history and should not be done on public commits!**

1. Create a new branch, and make a couple of commits on it.
2. Switch back to `main`, and make a couple of commits on it.
3. Inspect the situation with `git graph`.
4. Now rebase the new branch on top of `main` by first switching to the new branch, and then `git rebase main`.
5. Inspect again the situation with `git graph`. Notice that the commit hashes have changed - think about why!

### ✓ Solution

You will notice two things:

- History is now linear and does not contain merge commits.
- All the commit hashes that were on the branch that got rebased, have changed. This also demonstrates that `git rebase` is a command that alters history. The commit history looks as if the rebased commits were all done after the `main` commits.

## Tags

- A tag is a pointer to a commit but in contrast to a branch it **does not ever move** when creating new commits later.
- It can be useful to think of branches as sticky notes and of tags as [commemorative plaques](#).
- We use tags to record particular states or milestones of a project at a given point in time, like for instance versions (have a look at [semantic versioning](#), v1.0.3 is easier to understand and remember than 64441c1934def7d91ff0b66af0795749d5f1954a).
- There are two basic types of tags: annotated and lightweight.
- **Use annotated tags** since they contain the author and can be cryptographically signed using GPG, timestamped, and a message attached.

Let's add an annotated tag to our current state of the guacamole recipe:

```
$ git tag -a nobel-2023 -m "recipe I made for the 2023 Nobel banquet"
```

As you may have found out already, `git show` is a very versatile command. Try this:

```
$ git show nobel-2023
```

For more information about tags see for example [the Pro Git book](#) chapter on the subject.

---

## Summary

Let us pause for a moment and recapitulate what we have just learned:

```
$ git branch           # see where we are
$ git branch NAME      # create branch NAME
$ git switch NAME      # switch to branch NAME
$ git merge NAME        # merge branch NAME (to current branch)
$ git branch -d NAME    # delete branch NAME
$ git branch -D NAME    # delete unmerged branch NAME
```

Since the following command combo is so frequent:

```
$ git branch NAME      # create branch NAME
$ git switch NAME      # switch to branch NAME
```



There is a shortcut for it:

```
$ git switch --create NAME # create branch NAME and switch to it
```

## Typical workflows

With this there are two typical workflows:

```
$ git switch --create new-feature # create branch, switch to it
$ git commit                    # work, work, work, ..., and test
$ git switch main               # once feature is ready, switch to main
$ git merge new-feature         # merge work to main
$ git branch -d new-feature     # remove branch
```

Sometimes you have a wild idea which does not work. Or you want some throw-away branch for debugging:

```
$ git switch --create wild-idea # create branch, switch to it, work, work, work ...
$ git switch main              # realize it was a bad idea, back to main
$ git branch -D wild-idea      # it is gone, off to a new idea
```

### Branch-5: Test your understanding

Which of the following combos (one or more) creates a new branch and makes a commit to it?

1. 

```
$ git branch new-branch
$ git add file.txt
$ git commit
```

2. 

```
$ git add file.txt
$ git branch new-branch
$ git switch new-branch
$ git commit
```

3. 

```
$ git switch --create new-branch
$ git add file.txt
$ git commit
```

4. 

```
$ git switch new-branch
$ git add file.txt
$ git commit
```

### ✓ Solution

Both 2 and 3 would do the job. Note that in 2 we first stage the file, and then create the branch and commit to it. In 1 we create the branch but do not switch to it, while in 4 we don't give the `--create` flag to `git switch` to create the new branch.

### 📌 Keypoints

- A branch is a division unit of work, to be merged with other units of work.
- A tag is a pointer to a moment in the history of a project.

## Conflict resolution

### 📌 Objectives

- Understand merge conflicts sufficiently well to be able to fix them.

### Instructor note

- 20 min teaching/type-along
- 20 min exercise

## Conflicts in Git and why they are good

Imagine we start with the following text file:

```
1 tbsp cilantro
2 avocados
1 chili
1 lime
1 tsp salt
1/2 onion
```

On branch A somebody modifies:

```
2 tbsp cilantro
2 avocados
1 chili
2 lime
1 tsp salt
1/2 onion
```

On branch B somebody else modifies:

```
1/2 tbsp cilantro
2 avocados
1 chili
1 lime
1 tsp salt
1 onion
```

When we try to merge Git will figure out that we 2 limes and an entire onion but does not know whether to reduce or increase the amount of cilantro:

```
????????????????
2 avocados
1 chili
2 lime
1 tsp salt
1 onion
```

Git is very good at resolving modifications when merging branches and in most cases a `git merge` runs smooth and automatic. Then a merge commit appears (unless fast-forward; see [Optional exercises with branches](#)) without you even noticing.

But sometimes the **same portion** of the code/text is modified on two branches in **two different ways** and Git issues a **conflict**. Then you need to tell Git which version to keep (**resolve** it).

There are several ways to do that as we will see.

Please remember:

- It is good that Git conflicts exist: Git will not silently overwrite one of two differing modifications.
- Conflicts may look scary, but are not that bad after a little bit of practice. Also they are luckily rare.
- Don't be afraid of Git because of conflicts. You may not meet some conflicts using other systems because you simply can't do the kinds of things you do in Git.

- You can take human measures to reduce them.

### The human side of conflicts

- What does it mean if two people do the same thing in two different ways?
- What if you work on the same file but do two different things in the different sections?
- What if you do something, don't tell someone for 6 months, and then try to combine it with other people's work?
- How are conflicts avoided in other work? (Only one person working at once? Declaring what you are doing before you start, if there is any chance someone else might do the same thing, helps.)
- Minor conflicts (two people revise spelling) vs semantic (two people rewrite a function to add two different new features). How did Git solve these in branching/merging easily?

Now we can go to show how Git controls when there is actually a conflict.

## Preparing a conflict

### Instructor note

We do the following together as type-along/demo.

### If you got stuck previously or joined later

If you got stuck previously or joined later, you can apply the commands below. But **skip** this box if you managed to create branches.

```
$ cd .. # step out of the current directory

$ git clone https://github.com/coderefinery/recipe-before-merge.git
$ cd recipe-before-merge

$ git remote remove origin

$ git graph
```

Or call a helper to un-stuck it for you.

We will make two branches, make two conflicting changes (both increase and decrease the amount of cilantro), and later we will try to merge them together.

- Create two branches from `main`: one called `like-cilantro`, one called `dislike-cilantro`:

```
$ git branch like-cilantro main
$ git branch dislike-cilantro main
```

- On the two branches make **different modifications** to the amount of the **same ingredient**:
- On the branch `like-cilantro` we have the following change:

```
$ git diff main like-cilantro
```

```
diff --git a/ingredients.txt b/ingredients.txt
index e83294b..6cacd50 100644
--- a/ingredients.txt
+++ b/ingredients.txt
@@ -1,4 +1,4 @@
-* 1 tbsp cilantro
+* 2 tbsp cilantro
 * 2 avocados
 * 1 chili
 * 1 lime
```

- And on the branch `dislike-cilantro` we have the following change:

```
$ git diff main dislike-cilantro
```

```
diff --git a/ingredients.txt b/ingredients.txt
index e83294b..6484462 100644
--- a/ingredients.txt
+++ b/ingredients.txt
@@ -1,4 +1,4 @@
-* 1 tbsp cilantro
+* 1/2 tbsp cilantro
 * 2 avocados
 * 1 chili
 * 1 lime
```

## Merging conflicting changes

What do you expect will happen when we try to merge these two branches into main?

### Note

In case `git switch` does not work, your Git version might be older than from 2019. On older Git it is `git checkout` instead of `git switch`.

The first merge will work:

```
$ git switch main
$ git status
$ git merge like-cilantro

Updating 4e03d4b..3caa632
Fast-forward
 ingredients.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

But the second will fail:

```
$ git merge dislike-cilantro

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

Without conflict Git would have automatically created a merge commit, but since there is a conflict, Git did not commit:

```
$ git status

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   ingredients.txt

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

Git won't decide which to take and we need to decide. Observe how Git gives us clear instructions on how to move forward.

Let us inspect the conflicting file:

```
$ cat ingredients.txt

<<<<<< HEAD
* 2 tbsp cilantro
=====
* 1/2 tbsp cilantro
>>>>>> dislike-cilantro
* 2 avocados
* 1 chili
* 1 lime
* 1 tsp salt
* 1/2 onion
```

Git inserted resolution markers (the `<<<<<<`, `>>>>>>`, and `=====`).

Try also `git diff`:

```
$ git diff
```

```
diff --cc ingredients.txt
index 6cacd50,6484462..0000000
--- a/ingredients.txt
+++ b/ingredients.txt
@@@ -1,4 -1,4 +1,10 @@@
++<<<<<< HEAD
+* 2 tbsp cilantro
++=====
+ * 1/2 tbsp cilantro
++>>>>>> dislike-cilantro
+  * 2 avocados
+  * 1 chili
+  * 1 lime
```

`git diff` now only shows the conflicting part, nothing else.

## Conflict resolution

```
<<<<<< HEAD
* 2 tbsp cilantro
=====
* 1/2 tbsp cilantro
>>>>>> dislike-cilantro
```

We have to edit the code/text between the resolution markers. You only have to care about what Git shows you: Git stages all files without conflicts and leaves the files with conflicts unstaged.

## 📌 Steps to resolve a conflict

- Check status with `git status` and `git diff`.
- Decide what you keep (the one, the other, or both or something else). Edit the file to do this.
  - Remove the resolution markers, if not already done.
  - The file(s) should now look exactly how you want them.
- Check status with `git status` and `git diff`.
- Tell Git that you have resolved the conflict with `git add ingredients.txt` (if you use the Emacs editor with a certain plugin the editor may stage the change for you after you have removed the conflict markers).
- Verify the result with `git status`.
- Finally commit the merge with only `git commit`. Everything is pre-filled.

## Exercise: Create and resolve a conflict

### 🔧 Conflict-1: Create another conflict and resolve

In this exercise, we repeat almost exactly what we did above with a different ingredient.

1. Create two branches before making any modifications.
2. Again modify some ingredient on both branches.
3. Merge one, merge the other and observe a conflict, resolve the conflict and commit the merge.
4. What happens if you apply the same modification on both branches?
5. If you create a branch `like-avocados`, commit a change, then from this branch create another branch `dislike-avocados`, commit again, and try to merge both branches into `main` you will not see a conflict. Can you explain, why it is different this time?

### ✓ Solution

4: No conflict in this case if the change is the same.

5: No conflict in this case since in Git history one change happened after the other. The two changes are related and linked by Git history and one is a Git ancestor of the other. Git will assume that since we applied one change after the other, we meant this. There is nothing to resolve.

## Optional exercises with conflict resolution

### 🔧 (optional) Conflict-2: Resolve a conflict when rebasing a branch

1. Create two branches where you anticipate a conflict.
2. Try to merge them and observe that indeed they conflict.
3. Abort the merge with `git merge --abort`.



4. What do you expect will happen if you rebase one branch on top of the other? Do you anticipate a conflict? Try it out.

### ✓ Solution

Yes, this will conflict. If it conflicts during a merge, it will also conflict during rebase but the conflict resolution looks slightly different: You still need to look for conflict markers but you tell Git that you resolved a conflict with `git add` and then you continue with `git rebase --continue`. Follow instructions that you get from the Git command line.

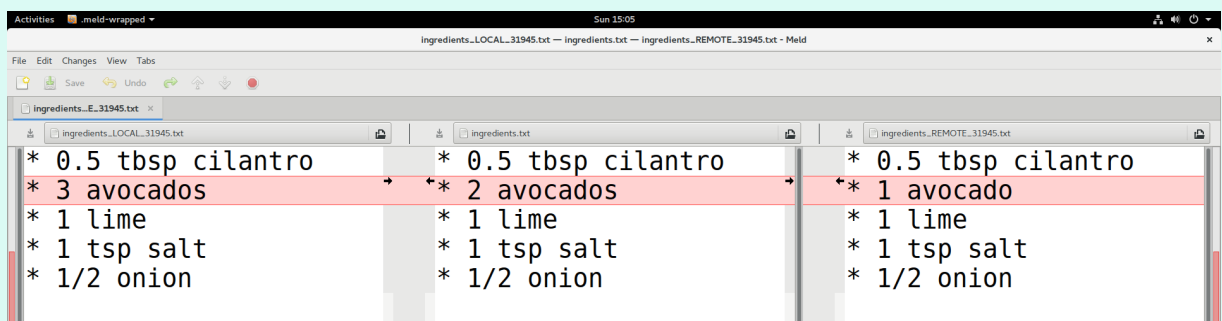
### 🔧 (optional) Conflict-3: Resolve a conflict using mergetool

- Again create a conflict (for instance disagree on the number of avocados).
- Stop at this stage:

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

- Instead of resolving the conflict manually, use a visual tool (requires installing one of the [visual diff tools](#)):

```
$ git mergetool
```



- Your current branch is left, the branch you merge is right, result is in the middle.
- After you are done, close and commit, `git add` is not needed when using `git mergetool`.

If you have not instructed Git to avoid creating backups when using mergetool, then to be on the safe side there will be additional temporary files created. To remove those you can do a git clean after the merging.

To view what will be removed:

```
$ git clean -n
```

To remove:

```
$ git clean -f
```

To configure Git to avoid creating backups at all:

```
$ git config --global mergetool.keepBackup false
```

---

## Using “ours” or “theirs” strategy

- Sometimes you know that you want to keep “ours” version (version on the branch you are on) or “theirs” (version on the merged branch).
- Then you do not have to resolve conflicts manually.
- See [merge strategies](#).

Example (merge and in doubt take the changes from current branch):

```
$ git merge -s recursive -Xours less-avocados
```

Or (merge and in doubt take the changes from less-avocados branch):

```
$ git merge -s recursive -Xtheirs less-avocados
```

---

## Aborting a conflicting merge

Sometimes you get a merge conflict but realize that you can’t solve it without talking to a colleague (who created the other change) first. What to do?

You can abort the merge and postponing conflict resolution by resetting the repository to `HEAD` (last committed state):

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

The repository looks then exactly as it was before the merge.

## Avoiding conflicts

- Human measures
  - Think and plan to which branch you will commit to.
  - Do not put unrelated changes on the same branch.
- Collaboration measures
  - Open an issue and discuss with collaborators before starting a long-living branch.
- Project layout measures
  - Modifying global data often causes conflicts.
  - Modular programming reduces this risk.
- Technical measures
  - **Share your changes early and often** - this is one of the happy, rare circumstances when everyone doing the selfish thing (e.g. `git push` as early as practical) results in best case for everyone!
  - Pull/rebase often to keep up to date with upstream.
  - Resolve conflicts early.

### Discussion

Discuss how Git handles conflicts compared to services like Google Drive.

### Keypoints

- Conflicts often appear because of not enough communication or not optimal branching strategy.

## Sharing repositories online

### Objectives

- We get a feeling for remote repositories ([more later](#)).
- We are able to publish a repository on the web.
- We are able to fetch and track a repository from the web.

### Instructor note

- 15 min demonstration/type-along
- 25 min exercise

## From our laptops to the web

We have seen that **creating Git repositories and moving them around is relatively simple** and that is great.

So far, if you only worked in the command line, everything was local and all snapshots, branches, and tags are saved under `.git`.

If we remove `.git`, we remove all Git history of a project.

### Discussion

- What if the hard disk fails?
- What if somebody steals my laptop?
- How can we collaborate with others across the web?

## Remotes

We will learn how to work with remote repositories in detail in the [collaborative distributed version control](#) lesson.

In this section we only want to get a taste to prepare us for other lessons where we will employ GitHub. Our goal is to publish our exercise guacamole recipe which we prepared in the previous episodes on the web. Don't worry, you will be able to remove it afterwards.

### If you don't have the recipe repository from previous episodes

Maybe you joined the workshop later or got stuck somewhere? **No problem!**

**If you don't have the recipe repository from previous episodes**, you can clone our version of the repository using (please **skip this** if you have the recipe repository on your computer already):

```
$ git clone https://github.com/coderefinery/recipe-before-merge.git recipe
$ cd recipe
$ git remote remove origin
```

Now you have a repository called `recipe` on your computer with a couple of commits. Further down we will also clarify what `git clone` does.

To store your git data on another server, you use **remotes**. A remote is a repository on its own, with its own branches We can **push** changes to the remote and **pull** from the remote.

You might use remotes to:

- Back up your own work.
- To collaborate with other people.

There are different types of remotes:

- If you have a server you can ssh to, you can use that as a remote.
- [GitHub](#) is a popular, closed-source commercial site.
- [GitLab](#) is a popular, open-core commercial site. Many universities have their own private GitLab servers set up.
- [Bitbucket](#) is yet another popular commercial site.
- Another option is [NotABug](#).
- We also operate a [Nordic research software repository platform](#). This is GitLab, free for researchers and allows private, cross-university sharing.

---

## Authenticating to GitHub: SSH or HTTPS?

How does Github know who you are? This is hard and there are two options.

- **SSH** is the classic method, using Secure Shell remote connection keys.
- **HTTPS** works with the **Git Credential Manager**, which is an extra add-on that works easily in Windows and Mac.

Read how to install them from the [installation instructions](#).

Test which one you should use:

**SSH**

**HTTPS**

Try this command:

```
$ ssh -T git@github.com
```

If it returns `Hi USERNAME! You've successfully authenticated, ...`, then SSH is configured and the following steps will work with the SSH cloning.

See our [installation instructions](#) to set up SSH access.

From now on, if you know that SSH works, you should always select SSH as the clone URL from GitHub, or translate the URL to start with the right thing yourself:

`git@github.com:` (with the trailing `:`).

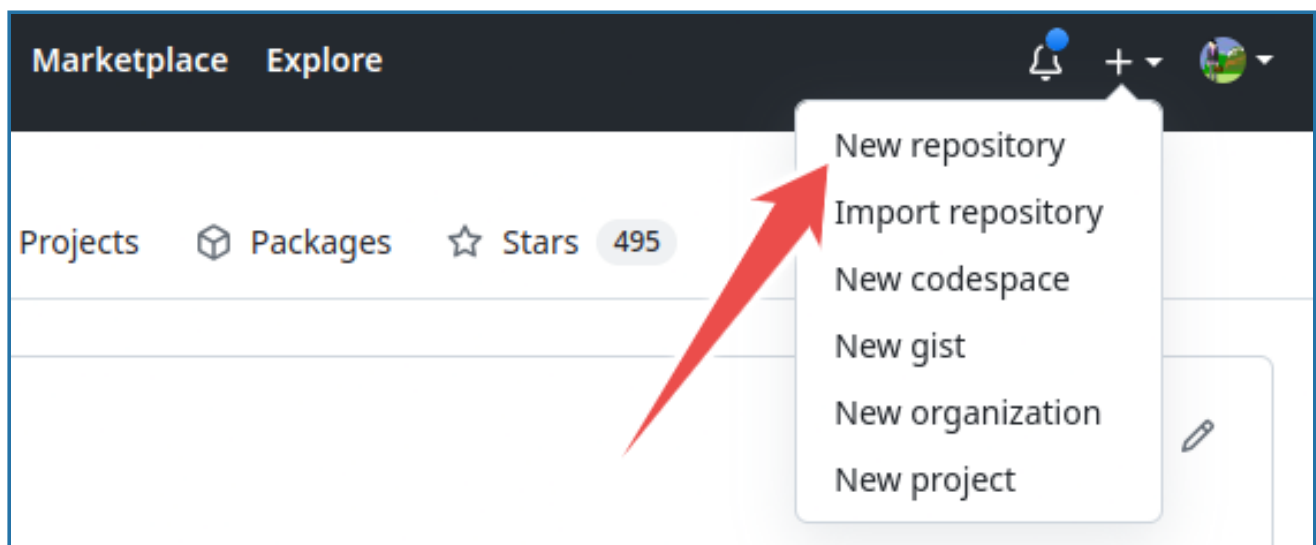
If you do not have these configured, please watch as we do this episode and you can check the [installation instructions](#) before the next [collaborative Git lesson](#), where we will need one of these set up.

## Publishing an existing repository from laptop to GitHub

### ❗ If you started in the browser and have nothing on your laptop yet


It is possible, that you already have a repository on GitHub if you followed the examples and exercises in previous episodes in the browser. In this case, please watch others publish their repository and try to clone your repository to the laptop using instructions at the bottom of this page.

First log into GitHub, then follow the screenshots and descriptions below.



Click on the “plus” symbol on top right, then on “New repository”.

Another way to create a new repository is to visit <https://github.com/new> directly.


Owner \*  bast / Repository name \*


✓ recipe is available

Great repository names are short and memorable. Need inspiration? How about [curly-octo-robot](#)?

Description (optional)

---

☒  **Public**  
Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**  
You choose who can see and commit to this repository.

---

**Initialize this repository with:**

☐ **Add a README file**  
This is where you can write a long description for your project. [Learn more about READMEs.](#)


**Add .gitignore**

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

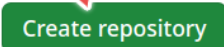
**Choose a license**

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

---

 You are creating a public repository in your personal account.

---



Choose a repository name, add a short description, but please **do not check** “Add a README file”\*\*. For “Add .gitignore” and “Choose a license” also leave as “None”. Finally “Create repository”.

Once you click the green “Create repository”, you will see a page similar to:

### Quick setup — if you've done this kind of thing before

or

HTTPS

SSH

git@github.com:bast/recipe.git

Get started by [creating a new repository](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

#### ...or create a new repository on the command line

```
echo "# recipe" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:bast/recipe.git
git push -u origin main
```

#### ...or push an existing repository from the command line

```
git remote add origin git@github.com:bast/recipe.git
git branch -M main
git push -u origin main
```

#### ...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

What this means is that we have now an empty project with either an HTTPS or an SSH address: click on the HTTPS and SSH buttons to see what happens.

We now want to follow the “... or push an existing repository from the command line:

1. Now go to your guacamole repository on your computer.
2. Check that you are in the right place with `git status`.
3. Copy paste the three lines to the terminal and execute those, in my case (you need to replace the “USER” part and possibly also the repository name):

SSH

HTTPS

See above for if SSH is the right option for you.

```
$ git remote add origin git@github.com:USER/recipe.git
```

Then:



```
$ git branch -M main
$ git push -u origin main
```

The meaning of the above lines:

- Add a remote reference with the name “origin”
- Rename current branch to “main”
- Push branch “main” to “origin”

You should now see:

```
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 221 bytes | 221.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:USER/recipe.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

**Reload your GitHub project website** and - taa-daa - your commits should now be online!  
What just happened? **Think of publishing a repository as uploading the `.git` part online.**

## ❗ Troubleshooting

**error: remote origin already exists.**

- Explanation: You probably ran a `git remote add origin ...` command, then changed your mind about HTTPS or SSH and then tried to run the other `git remote add origin ...` command but “origin” then already exists.
- Recovery:
  - First remove “origin” with `git remote remove origin`
  - Then run the correct `git remote add origin ...` command

**remote contains work that you do not have**

- Explanation: You probably clicked on “Add a README file” and now the repository on GitHub is not empty but contains one commit and locally you have a different history. Git now prevents you from accidentally overwriting the history on GitHub.
- Recovery:
  - Use `git push --force` instead of `git push`, which will force Git to overwrite the history on GitHub

- Note that this is a powerful but also possibly dangerous option but here it helps us. If it's a brand new repo, it probably is fine to do this. For real repositories, don't do this unless you are very sure what is happening.

## Cloning a repository from GitHub to laptop

Now other people can clone this repository and contribute changes. In the [collaborative distributed version control](#) lesson we will learn how this works.

At this point only a brief demo - if you copy the SSH or HTTPS address, you can clone repositories like this (again adapt the `USER` in the "USER/repository.git" part):

SSH

HTTPS

```
$ git clone git@github.com:USER/recipe.git
```

From now on, if you are using SSH, you should pay attention and make sure your clone URLs start with `git@github.com:` now and in future lessons.

This creates a directory called "recipe" unless it already exists. You can also specify the target directory on your computer:

SSH

HTTPS

```
$ git clone git@github.com:USER/recipe.git myrecipe
```

What just happened? **Think of cloning as downloading the `.git` part to your computer.** After downloading the `.git` part, the default branch (the branch pointed to by HEAD) is automatically checked out.

### ! Keypoints

- A repository can have one or multiple remotes (we will revisit these later).
- Local branches often track remote branches.
- A remote serves as a full backup of your work.

- We'll properly learn how to use these in the [collaborative distributed version control](#).

## Inspecting history

### 📌 Objectives

- Be able find a line of code, find out why it was introduced and when.
- Be able to quickly find the commit that changed a behavior.

### Instructor note

- 30 min teaching/type-along
- 30 min exercise

### ⚙️ Preparation

Please make sure that you do not clone repositories inside an already tracked folder:

```
$ git status
```

If you are inside an existing Git repository, step out of it. You need to find a different location since we will clone a new repository.

If you see this message, this is good in this case:

```
fatal: not a git repository (or any of the parent directories): .git
```

## Our toolbox for history inspection

### Instructor note

First the instructor demonstrates few commands on a real life example repository <https://github.com/networkx/networkx> (mentioned in the amazing site [The Programming Historian](#)). Later we will practice these in an archaeology exercise (below).

## Warm-up: “Git History” browser

As a warm-up we can try the “[Git History](#)” browser on the README.rst file of the [networkx](#) repository:

- Visit and browse <https://github.githistory.xyz/networkx/networkx/blob/main/README.rst> (use left/right keys).
- You can try this on some of your GitHub repositories, too!

## Searching text patterns in the repository

### Command line

### Browser (GitHub)

With `git grep` you can find all lines in a repository which contain some string or regular expression. This is useful to find out where in the code some variable is used or some error message printed:

```
$ git grep TEXT
$ git grep "some text with spaces"
```

In the [networkx](#) repository you can try:

```
$ git clone https://github.com/networkx/networkx
$ cd networkx
$ git grep -i fixme
```

While `git grep` searches the **current state** of the repository, it is also possible to search through all changes with `git log -S sometext` which can be useful to find where something got removed.

## Inspecting individual commits

### Command line

### Browser (GitHub)

We have seen this one before already. Using `git show` we can inspect an individual commit if we know its hash:

```
$ git show HASH
```

For instance:

```
$ git show 759d589bdfa61aff99e0535938f14f67b01c83f7
```

## Line-by-line code annotation with metadata

With `git annotate` you can see line by line who and **when** the line was modified last. It also prints the precise hash of the last change which modified each line. Incredibly useful for reproducibility.

### Command line

### Browser (GitHub)

```
$ git annotate FILE
```

Example:

```
$ git annotate networkx/convert_matrix.py
```

If you annotate in a terminal and the file is longer than the screen, Git by default uses the program `less` to scroll the output. Use `/sometext <ENTER>` to find “sometext” and you can cycle through the results with `n` (next) and `N` (last). You can also use page up/down to scroll. You can quit with `q`.

## Discussion

Discuss how these relatively trivial changes affect the annotation:

- Wrapping long lines of text/code into shorter lines
- Auto-formatting tools such as `black`
- Editors that automatically remove trailing whitespace

## Inspecting code in the past

### Command line

### Browser (GitHub)

We can create branches pointing to a commit in the past. This is the recommended mechanism to inspect old code:

```
$ git switch --create BRANCHNAME HASH
```

Example (lines starting with “#” are only comments):

```
$ # create branch called "older-code" from hash 347e6292419b
$ git switch --create older-code 347e6292419bd0e4bff077fe971f983932d7a0e9

$ # now you can navigate and inspect the code as it was back then
$ # ...

$ # after we are done we can switch back to "main"
$ git switch main

$ # if we like we can delete the "older-code" branch
$ git branch -d older-code
```

On old Git versions which do not know the `switch` command (before 2.23), you need to use this instead:

```
$ git checkout -b BRANCHNAME SOMEHASH
```

## Exercise: Basic archaeology commands

### History-1: Explore basic archaeology commands

Let us explore the value of these commands in an exercise. Future exercises do not depend on this, so it is OK if you do not complete it fully.

Exercise steps:

- Clone this repository: <https://github.com/networkx/networkx.git>. Then step into the new directory and create an exercise branch from the networkx-2.6.3 tag/release:

```
$ git clone https://github.com/networkx/networkx.git
$ cd networkx
$ git switch --create exercise networkx-2.6.3
```

On old Git versions which do not know the `switch` command (before 2.23), you need to use this instead:

```
$ git checkout -b exercise networkx-2.6.3
```

Then using the above toolbox try to:

1. Find the code line which contains `"Logic error in degree_correlation"`.
2. Find out when this line was last modified or added. Find the actual commit which modified that line.
3. Inspect that commit with `git show`.
4. Create a branch pointing to the past when that commit was created to be able to browse and use the code as it was back then.
5. How would you bring the code to the version of the code right before that line was last modified?

## ✓ Solution

We provide here a solution for the command line but we also encourage you to try to solve this in the browser.

1. We use `git grep`:

```
$ git grep "Logic error in degree_correlation"
```

This gives the output:

```
networkx/algorithms/threshold.py:                print("Logic error in  
degree_correlation", i, rdi)
```

Maybe you also want to know the line number:

```
$ git grep -n "Logic error in degree_correlation"
```

2. We use `git annotate`:

```
$ git annotate networkx/algorithms/threshold.py
```

Then search for “Logic error” by typing `/Logic error` followed by Enter. The last commit that modified it was `90544b4fa` (unless that line changed since).

3. We use `git show`:

```
$ git show 90544b4fa
```

4. Create a branch pointing to that commit (here we called the branch “past-code”):

```
$ git branch past-code 90544b4fa
```

5. This is a compact way to access the first parent of `90544b4fa` (here we called the branch “just-before”):

```
$ git switch --create just-before 90544b4fa~1
```

## Finding out when something broke/changed with `git bisect`

*“But I am sure it used to work! Strange.”*

Sometimes you realize that something broke. You know that it used to work. You do not know when it broke.

### How would you solve this?

Before we go on first discuss how you would solve this problem: You know that it worked 500 commits ago but it does not work now.

- How would you find the commit which changed it?
- Why could it be useful to know the commit that changed it?

We will probably arrive at a solution which is similar to `git bisect`:

- First find out a commit in past when it worked.

```
$ git bisect start
$ git bisect good f0ea950 # this is a commit that worked
$ git bisect bad main     # last commit is broken
```

- Now compile and/or run and/or test and decide whether “good” or “bad”.
- This is how you can tell Git that this was a working commit:

```
$ git bisect good
```

- And this is how you can tell Git that this was not a working commit:

```
$ git bisect bad
```



- Then bisect/iterate your way until you find the commit that broke it.
- If you want to go back to start, type `git bisect reset`.
- This can even be automatized with `git bisect run SCRIPT`. For this you write a script that returns zero/non-zero (success/failure).

## Optional exercise: Git bisect

### (optional) History-2: Use git bisect to find the bad commit

In this exercise, we use `git bisect` on an example repository. It is OK if you do not complete this exercise fully.

Begin by cloning <https://github.com/coderefinery/git-bisect-exercise>.

#### Motivation

The motivation for this exercise is to be able to do archaeology with Git on a source code where the bug is difficult to see visually. **Finding the offending commit is often more than half the debugging.**

#### Background

The script `get_pi.py` approximates pi using terms of the Nilakantha series. It should produce 3.14 but it does not. The script broke at some point and produces 3.57 using the last commit:

```
$ python get_pi.py  
  
3.57
```

At some point within the 500 first commits, an error was introduced. The only thing we know is that the first commit worked correctly.

#### Your task

- Clone this repository and use `git bisect` to find the commit which broke the computation ([solution - spoiler alert!](#)).
- Once you have found the offending commit, also practice navigating to the last good commit.
- Bonus exercise: Write a script that checks for a correct result and use `git bisect run` to find the offending commit automatically ([solution - spoiler alert!](#)).

#### Hints

Finding the first commit:

```
$ git log --oneline | tail -n 1
```

How to navigate to the parent of a commit with hash SOMEHASH:

```
$ git switch --create BRANCHNAME SOMEHASH~1
```

Instead of a tilde you can also use this:

```
$ git switch --create BRANCHNAME SOMEHASH^
```

### ! Keypoints

- git log/grep/annotate/show/bisect is a powerful combination when doing archaeology in a project.
- `git switch --create NAME HASH` is the recommended mechanism to inspect old code.

## Practical advice: how much Git is necessary?

### Instructor note

- 20 min teaching/discussion

## What level of branching complexity is necessary for each project?

### Simple personal projects

- Typically start with just the `main` branch.
- Use branches for unfinished/untested ideas.
- Use branches when you are not sure about a change.
- Use tags to mark important milestones.

### Projects with few persons: you accept things breaking sometimes

- It might be reasonable to commit to the `main` branch and feature branches.

### Projects with few persons: changes are reviewed by others

- You create new feature branches for changes.
- Changes are reviewed before they are merged to the `main` branch (more about that in the [collaborative Git lesson](#)).
- The `main` branch is write-protected and can only be changed with pull requests or merge requests.

## When you distribute releases

- If you want to patch releases, you probably need release branches.
- The `main` branch and release branches are read-only.
- Many [branching models](#) exist.

---

## How about staging and committing?

- It is OK to start committing directly by doing `git commit SOMEFILE`.
- Commit early and often: rather create too many commits than too few. You can always combine commits later.
- Once you commit, it is very, very hard to really lose your code.
- Always fully commit (or stash) before you do dangerous things, so that you know you are safe. Otherwise it can be hard to recover.
- Later you can start using the staging area.
- Later start using `git add -p` and/or `git commit -p`.

---

## How large should a commit be?

- Better too small than too large (easier to combine than to split).
- Often I make a commit at the end of the day (this is a unit I would not like to lose).
- Smaller sized commits may be easier to review for others than huge commits.
- Imperfect commits are better than no commits.
- A commit should not contain unrelated changes to simplify review and possible repair/adjustments/undo later (but again: imperfect commits are better than no commits).

---

### Keypoints

- There is no one size fits all - start simple and grow your project.

### Discussion

How do you [plan to] use Git?

- Advanced users or beginners, please provide your input in the online collaborative document.

# What to avoid

The idea for this page came up during a discussion: quick reference page of things that you should not do with Git. Basically, a list of some common Git gotchas.

**Postponing commits because the changes are “unfinished”/“ugly”:** It is better to have many OK-ish commits than too few perfect commits. Too few perfect commits are probably too large and make undoing more difficult. It is also easier to combine too small commits than it is to split too large commits. The longer you wait because of this, the harder it will be to commit it at all.

**Not updating your branch before starting new work:** Few things are worse than doing work and then noticing a lot of conflicts because unrelated but conflicting work was done in the meantime or even before you started. Or noticing that someone else has already done it. This problem is largest when you come back to an active project weeks or months later.

**Commit unrelated changes together:** Makes it difficult to undo changes since it can undo also the unrelated change. But, this is the counterpoint to the first point: In the end, it is probably better to make big ugly commits than to never commit. This is especially true in the chaotic early phases of small projects.

**Too ambitious branch which risks to never get completed:** The branch will never merge back and be so big and so ambitious with too many/big features that the risk is high that once it is really ready, there are conflicts everywhere.

**Committing generated files:** See [Ignoring files and paths with .gitignore](#).

**Over-engineering the branch layout and safeguards in small projects:** This may prevent people from contributing (maybe even including yourself?). Add more restrictions and safeguards as the project and the group of collaborators grows.

**Commit messages that explain what has been changed but do not explain why it has been changed:** This is as useful as code comments which describe the “obvious” such as “this is a loop” instead of explaining why something is done this way. But don’t let perfect commit messages stop you from the most important point, committing often (first point).

**Commit huge files:** Huge files get sometimes accidentally added and committed which can significantly increase the repository size. Note that a subsequent `git rm` does not remove the addition from the repository history. Code review can help detecting accidental large file additions. You can also add an automated test which checks for file sizes during a pull request or merge request.

Question to all seasoned Git users: What are we missing on this page? Please contribute improvements.

## Using the Git staging area

### 📌 Objectives

- Learn how to tell a story with your commit history.
- Demystify the Git staging area.

### Instructor note

- 10 min teaching/type-along
- 10 min exercise

## Commit history is telling a story

Your *current code* is very important, but the *history* can be just as important - it tells a story about how your code came to be.

- Each individual line of code rarely stands alone.
- You often want to see all the related changes together.
- But you also hardly ever do one thing at once.

Along with your code, Git creates a *history* for you, and if your history is clear then you are a long way to organized code.

### 💬 Discussion

Here are five types of history. What are the advantages and disadvantages of each, when you look at it later?

#### Example 1:

```
b135ec8 add features A, B, and C
```

#### Example 2 (newest commit is on top):

```
6f0d49f implement feature C
fee1807 implement feature B
6fe2f23 implement feature A
```

### Example 3:

```
ab990f4 saving three months of miscellaneous work I forgot to commit
```

### Example 4 (newest commit is on top):

```
bf39f9d more work on feature B
45831a5 removing debug prints for feature A and add new file
bddb280 more work on feature B and make feature A compile again
72d78e7 feature A did not work and started work on feature B
b135ec8 now feature A should work
72e0211 another fix to make it compile
61dd3a3 forgot file and bugfix
49dc419 wip (work in progress)
```

### Example 5 (newest commit is on top):

```
1949dc4 Work of 2020-04-07
a361dd3 Work of 2020-04-06
1172e02 Work of 2020-04-03
e772d78 Work of 2020-04-02
```

Discuss these examples. Can you anticipate problems?

We want to have nice commits. But we also want to “save often” (checkpointing) - how can we have both?

- We will now learn to create nice commits using `git commit --patch` and/or the staging area.
- Staging addresses the issue of having unrelated changes in the same commit or having one logical change spread over several commits.
- The staging area isn't the only way to organize your history nicely, some alternatives are discussed at the end of the lesson.

## Interactive commits

- The simplest ways to solve this is to do **interactive commits**: the `git commit --patch` option (or `git commit -p` for short).
- It will present you with every change you have made individually, and you can decide which ones to commit right now.
- Reference and key commands
  - `git commit --patch` to start the interactive commit

- `y` to use the change
- `n` to skip the change
- `s` (split) if there are several changes grouped together, but separated by a blank line, split them into separate choices.
- `q` aborts everything.
- `?` for more options.
- The `-p` option is also available on `commit`, `restore`, `checkout`, `reset`, and `add`.

## Exercise: Interactive commits

### Staging-1: Perform an interactive commit

One option to help us create nice logical commits is to stage *interactively* with `git commit --patch`:

1. Make two changes in `instructions.txt`, at the top and bottom of the file. **Make sure that they are separated by at least several unmodified lines.**
2. Run `git commit --patch`. Using the keystrokes above, commit one of the changes.
3. Do it again for the other change.
4. When you're done, inspect the situation with `git log`, `git status`, `git diff` and `git diff --staged`.
5. When would this be useful?

### ✓ Solution

This can be useful if you have several modification in a file (or several files) but you decide that it would be beneficial to save them as two (or more) separate commits.

## The staging area

- The interactive commits above are great, but what if there are so many changes that you can't sort them out in one shot?
- What if you make progress and want to record it somehow, but it's not ready to be committed?
- The **staging area** is a place to record things before committing.

### Instructor note

We give two examples and the instructor can pick one or both:

- Analogy using moving boxes
- Analogy using shopping receipts

### Discussion

## Analogy using moving boxes

- You're moving and you have a box to pack your things in.
- You can put stuff into the box, but you can also take stuff out of the box.
- You wouldn't want to mix items from the bathroom, kitchen, and living room into the same box.
- The box corresponds to the staging area of Git, where you can craft your commits.
- Committing is like sealing the box and sticking a label on it.
- You wouldn't want to label your box with "stuff", but rather give a more descriptive label.
- See also <https://dev.to/sublimegeek/git-staging-area-explained-like-im-five-1anh>

## Analogy using shopping receipts

- You need to go shopping and buy some stuff for work and for home. You need two separate receipts.
- Bad idea: go through the store get home stuff, pay, start at the beginning and go through the store again. This is inefficient and annoying.
- What you actually do:
  - Go through the store and put everything you need in your shopping basket.
  - Get to the check-out. Put your home stuff on the conveyor belt ( `git add` ). Check both the belt ( `git diff --staged` ) and your basket ( `git diff` ) to make sure you got all your home stuff.
  - Pay ( `git commit` )
  - Repeat for work stuff.

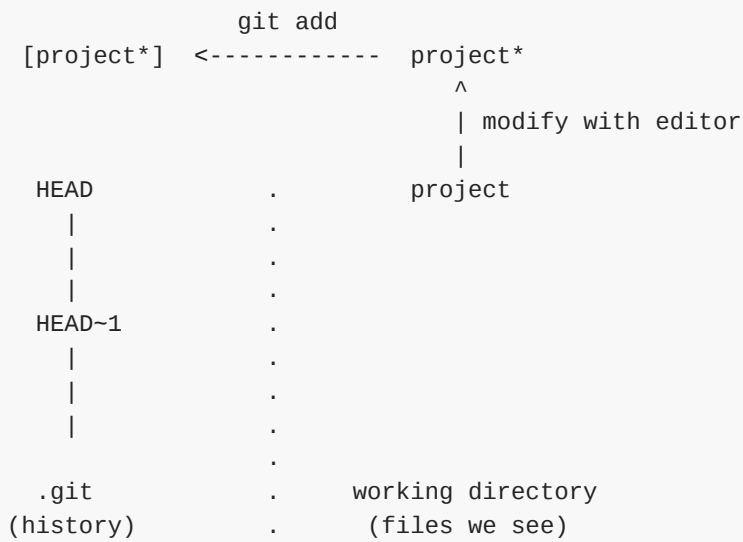
In order to keep organized, you have to use multiple locations to stage things in sequence.

## Staging area commands

The staging area is a middle ground between what you have done to your files (the **working directory**) and what you have last committed (the **HEAD commit**). Just like the name implies, it lets you prepare (**stage**) what the next commit will be and most importantly give you tools to easily know what is going on. This adds some complexity but also adds more flexibility to selectively prepare commits since **you can modify and stage several times before committing**.

**git add** stages/prepares for the next commit:





Going back to the last staged version:

```
git restore
[project*] -----> project*
```

```
HEAD      .
|         .
|         .
|         .
HEAD~1    .
|         .
|         .
|         .
|         .
.git      .      working directory
(history) .      (files we see)
```

Unstaging changes with **git restore --staged**:

```
git restore --staged
-----> project*
```

```
HEAD      .
|         .
|         .
|         .
HEAD~1    .
|         .
|         .
|         .
|         .
.git      .      working directory
(history) .      (files we see)
```

Discarding unstaged changes:

```
project*
|
| git restore
v
project

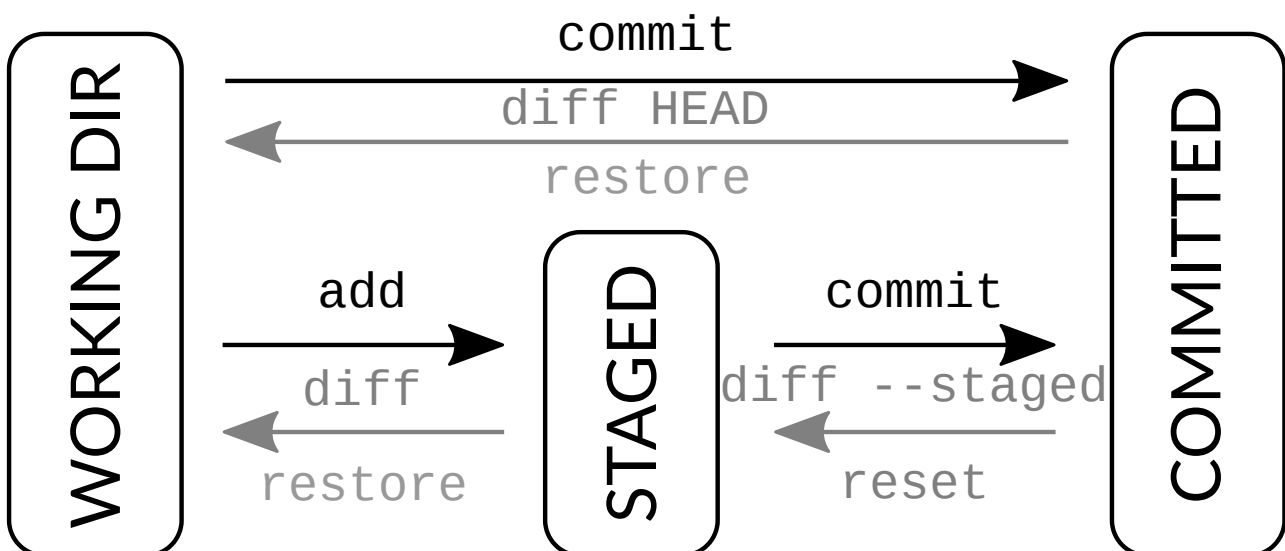
HEAD      .
|         .
|         .
|         .
HEAD~1    .
|         .
|         .
|         .
|         .
.git      .      working directory
(history) .      (files we see)
```

Comparing:

```

git diff
[project*] <-----> project*
  ^                      ^
  | git diff --staged    | git diff HEAD
  v                      v
HEAD                      .
|                          .
|                          .
|                          .
HEAD~1                     .
|                          .
|                          .
|                          .
|                          .
.git                       .      working directory
(history)                  .      (files we see)

```



*The different states of the repository and the commands to move from one to another.*

## Exercise: Using the staging area

## 👉 Staging-2: Use the staging area to make a commit in two steps

1. In your recipe example, make two different changes to `ingredients.txt` and `instructions.txt` which do not go together.
2. Use `git add` to stage one of the changes.
3. Use `git status` to see what's going on, and use `git diff` and `git diff --staged` to see the changes.
4. Feel some regret and unstage the staged change.

## Discussion

- When is it better to “save” a change as commit, when is it better to “save” it with `git add`?
- Is it a problem to commit many small changes?

### 📌 Keypoints

- The staging area helps us to create well-defined commits.

## Undoing and recovering

### 📌 Objectives

- Learn to undo changes safely
- See when undone changes are permanently deleted and when they can be retrieved

### Instructor note

- 25 min teaching/type-along
- 25 min exercise

One of the main points of version control is that you can *go back in time to recover*. Unlike this xkcd comic implies: <https://xkcd.com/1597/>

In this episode we show a couple of commands that can be used to undo mistakes. We also list a couple of common mistakes and discuss how to recover from them. Some commands preserve the commit history and some modify commit history. Modifying history? Isn't a “commit” permanent?

- You can modify old commit history.
- But if you have shared that history already, *modifying it can make a huge mess*.

### 📌 It is almost always possible to recover

As long as you commit something once (or at least `git add` it), you can almost always go back to it, no matter what you do. But you may need to ask Stack Overflow or your local guru... *until that guru becomes you*.

### 📌 Nice resource to visually simulate Git operation

[git-sim](#) is a nice resource to visually simulate Git operations listed in this episode below, in your own repos with a single terminal command.

## Undoing your recent, uncommitted and unstaged changes (preserves history)

### Note

In case `git restore` does not work, your Git version might be older than from 2019. On older Git it is `git checkout` instead of `git restore`.

You do some work, and want to **undo your uncommitted and unstaged modifications**. You can always do that with:

- `git restore .` (the dot means “here and in all folders below”)

You can also undo things selectively:

- `git restore -p` (decide which portions of changes to undo) or `git restore PATH` (decide which path/file)

If you have staged changes, you have at least two options to undo the staging:

- `git restore --staged .` followed by `git status` and `git restore .`
- `git reset --hard HEAD` throws away everything that is not in last commit (`HEAD` - this literal word, this isn't a placeholder)

---

## Reverting commits (preserves history)

Imagine we made a few commits. We realize that the latest commit `e02efcd` was a mistake and we wish to undo it:

```
$ git log --oneline
e02efcd (HEAD -> main) not sure this is a good idea
b4af65b improve the documentation
e7cf023 don't forget to enjoy
79161b6 add half an onion
a3394e3 adding README
3696246 adding instructions
f146d25 adding ingredients
```

A safe way to undo the commit is to revert the commit with `git revert`:

```
$ git revert e02efcd
```

This creates a **new commit** that does the opposite of the reverted commit. The old commit remains in the history:

```
$ git log --oneline

d3fc63a (HEAD -> main) Revert "not sure this is a good idea"
e02efcd not sure this is a good idea
b4af65b improve the documentation
e7cf023 don't forget to enjoy
79161b6 add half an onion
a3394e3 adding README
3696246 adding instructions
f146d25 adding ingredients
```

You can revert any commit, no matter how old it is. It doesn't affect other commits you have done since then - but if they touch the same code, you may get a conflict (which we'll learn about later).

## Exercise: Revert a commit

### Undoing-1: Revert a commit

- Create a commit (commit A).
- Revert the commit with `git revert` (commit B).
- Inspect the history with `git log --oneline`.
- Now try `git show` on both the reverted (commit A) and the newly created commit (commit B).

## Adding to the previous commit (modifies history)

Sometimes we commit but realize we forgot something. We can amend to the last commit:

```
$ git commit --amend
```

This can also be used to modify the last commit message.

Note that this **will change the commit hash**. This command **modifies the history**. This means that we avoid this command on commits that we have shared with others.

## Exercise: Modify a previous commit

### Undoing-2: Modify a previous commit

1. Make an incomplete change to the recipe or a typo in your change, `git add` and `git commit` the incomplete/unsatisfactory change.

2. Inspect the unsatisfactory but committed change with `git show`. Remember or write down the commit hash.
3. Now complete/fix the change but instead of creating a new commit, add the correction to the previous commit with `git add`, followed by `git commit --amend`. What changed?

### ✓ Solution

One thing that has changed now is the commit hash. Modifying the previous commit has changed the history. This is OK to do on commits that other people don't depend on yet.

## Rewinding branches (modifies history)

You can **reset branch history** to move your branch back to some point in the past.

- `git reset --hard HASH` will force a branch label to any other point. All other changes are lost (but it is possible to recover if you force reset by mistake).
- Be careful if you do this - it can mess stuff up. Use `git graph` a lot before and after.

### Exercise: Git reset

#### Undoing-3: Destroy our experimentation in this episode

After we have experimented with reverts and amending, let us destroy all of that and get our repositories to a similar state.

- First, we will look at our history ( `git log` / `git graph` ) and find the last commit `HASH` before our tests.
- Then, we will `git reset --hard HASH` to that.
- Then, `git graph` again to see what happened.

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

```
d3fc63a (HEAD -> main) Revert "not sure this is a good idea"  
e02efcd not sure this is a good idea  
b4af65b improve the documentation  
e7cf023 don't forget to enjoy  
79161b6 add half an onion  
a3394e3 adding README  
3696246 adding instructions  
f146d25 adding ingredients
```

```
$ git reset --hard b4af65b
```

```
HEAD is now at b4af65b improve the documentation
```

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

```
b4af65b (HEAD -> main) improve the documentation  
e7cf023 don't forget to enjoy  
79161b6 add half an onion  
a3394e3 adding README  
3696246 adding instructions  
f146d25 adding ingredients
```

---

## Recovering from committing to the wrong branch

It is easy to forget to create a branch or to create it and forget to switch to it when committing changes.

Here we assume that we made a couple of commits but we realize they went to the wrong branch.

### Solution 1 using git cherry-pick:

1. Make sure that the correct branch exists and if not, create it. Make sure to create it from the commit hash where you wish you had created it from: `git branch BRANCHNAME HASH`
2. Switch to the correct branch.
3. `git cherry-pick HASH` can be used to take a specific commit to the current branch. Cherry-pick all commits that should have gone to the correct branch, **from oldest to most recent**.
4. Rewind the branch that accidentally got wrong commits with `git reset --hard` (see also above).

**Solution 2 using `git reset --hard`** (makes sense if the correct branch should contain all commits of the accidentally modified branch):

1. Create the correct branch, pointing at the latest commit: `git branch BRANCHNAME`.
2. Check with `git log` or `git graph` that both branches point to the same, latest, commit.



3. Rewind the branch that accidentally got wrong commits with `git reset --hard` (see also above).

## Recovering from merging/pulling into the wrong branch

`git merge`, `git rebase`, and `git pull` modify the **current** branch, never the other branch. But sometimes we run this command on the wrong branch.

1. Check with `git log` the commit hash that you would like to rewind the wrongly modified branch to.
2. Rewind the branch that accidentally got wrong commits with `git reset --hard HASH` (see also above).

## Recovering from conflict after pulling changes

Pulling changes with `git pull` can create a conflict since `git pull` always also includes a `git merge` (more about this in the [collaborative Git lesson](#)).

The recovery is same as described in [Conflict resolution](#). Either resolve conflicts or abort the merge with `git merge --abort`.

### Undoing-4: Test your understanding

1. What happens if you accidentally remove a tracked file with `git rm`, is it gone forever?
2. Is it OK to modify commits that nobody has seen yet?
3. What situations would justify to modify the Git history and possibly remove commits?

#### ✓ Solution

1. It is not gone forever since `git rm` creates a new commit. You can revert the commit to get the file back.
2. If you haven't shared your commits with anyone it can be alright to modify them.
3. If you have shared your commits with others (e.g. pushed them to GitHub), only extraordinary conditions would justify modifying history. For example to remove sensitive or secret information.

## Interrupted work

### 📌 Objectives

- Learn to switch context or abort work without panicking.

## Instructor note

- 10 min teaching/type-along
- 15 min exercise

## Keypoints

- There is almost never reason to clone a fresh copy to complete a task that you have in mind.

## Frequent situation: interrupted work

We all wish that we could write beautiful perfect code. But the real world is much more chaotic:

- You are in the middle of a “Jackson-Pollock-style” debugging spree with 27 modified files and debugging prints everywhere.
- Your colleague comes in and wants you to fix/commit something right now.
- What to do?

Git provides lots of ways to switch tasks without ruining everything.

## Option 1: Stashing

The **stash** is the first and easiest place to temporarily “stash” things.

- `git stash` will put working directory and staging area changes away. Your code will be same as last commit.
- `git stash pop` will return to the state you were before. Can give it a list.
- `git stash list` will list the current stashes.
- `git stash save NAME` is like the first, but will give it a name. Useful if it might last a while.
- `git stash save [-p] [filename]` will stash certain files and/or by patches.
- `git stash drop` will drop the most recent stash (or whichever stash you give).
- The stashes form a stack, so you can stash several batches of modifications.

## Exercise: Stashing

### Interrupted-1: Stash some uncommitted work

1. Make a change.
2. Check status/diff, stash the change with `git stash`, check status/diff again.
3. Make a separate, unrelated change which doesn't touch the same lines. Commit this change.
4. Pop off the stash you saved with `git stash pop`, and check status/diff.

5. Optional: Do the same but stash twice. Also check `git stash list`. Can you pop the stashes in the opposite order?
6. Advanced: What happens if stashes conflict with other changes? Make a change and stash it. Modify the same line or one right above or below. Pop the stash back. Resolve the conflict. Note there is no extra commit.
7. Advanced: what does `git graph` show when you have something stashed?

### ✓ Solution

5: Yes you can. With `git stash pop INDEX` you can decide which stash index to pop.

6: In this case Git will ask us to resolve the conflict the same way when resolving conflicts between two branches.

7: It shows an additional commit hash with `refs/stash`.

## Option 2: Create branches

You can use branches almost like you have already been doing if you need to save some work. You need to do something else for a bit? Sounds like a good time to make a feature branch.

You basically know how to do this:

```
$ git switch --create temporary # create a branch and switch to it
$ git add PATHS                # stage changes
$ git commit                   # commit them
$ git switch main              # back to main, continue your work there ...
$ git switch temporary         # continue again on "temporary" where you left off
```

Later you can merge it to main or rebase it on top of main and resume work.

## Storing various junk you don't need but don't want to get rid of

It happens often that you do something and don't need it, but you don't want to lose it right away. You can use either of the above strategies to stash/branch it away: using branches is probably better because branches are less easily overlooked if you come back to the repository in few weeks. Note that if you try to use a branch after a long time, conflicts might get really bad but at least you have the data still.

## Aliases and configuration

### ! Objectives

- Learn to use aliases for most common commands.

Are you getting tired of typing so much? In Git you can define aliases (shortcuts):

- These are great because they can save you time typing.
- But it's easy to forget them, get confused, or be inconsistent with your colleagues.

There is plenty of other configuration for Git, that can make it nicer.

## Aliases

- Aliases offer a way to improve the usability of Git: for example `git ci` instead of `git commit`.
- Aliases are based on simple string replacement in the command.
- Aliases can either be specific to a repository or global.
  - Global aliases help you do the things you are used to across Git projects.
  - Per-project aliases can also be created.
- Global aliases are stored in `~/.gitconfig`.

## Example alias: git graph

A very useful shortcut which we use a lot in our workshops:

```
$ git config --global alias.graph "log --all --graph --decorate --oneline"
$ cd your_git_repository
$ git graph
```

## Using external commands

It is possible to call external commands using the exclamation mark character "!". In this example here we create a local alias which is stored in `.git/config` and not synchronized with remotes:

```
$ cd your_git_repository
$ git config alias.hi '!echo hello'
$ git hi
```

### Food for thought: When to alias?

- How many times should you wait before aliasing a command?
- Do you believe a list of generic two-letter acronyms for common commands will save your time?

# List of aliases the instructors use

You are welcome to reuse, suggest, improve. You can see your current aliases in

```
~/.gitconfig
```

```
$ git config --global alias.ap "add --patch"
$ git config --global alias.br branch
$ git config --global alias.ci "commit -v"
$ git config --global alias.cip "commit --patch -v"
$ git config --global alias.cl "clone --recursive"
$ git config --global alias.di diff
$ git config --global alias.dic "diff --staged --color-words"
$ git config --global alias.diw "diff --color-words"
$ git config --global alias.dis "!git --no-pager diff --stat"
$ git config --global alias.fe fetch
$ git config --global alias.graph "log --all --graph --decorate --oneline"
$ git config --global alias.rem remote
$ git config --global alias.st status
$ git config --global alias.su "submodule update --init --recursive"
```

Here is what they do:

- `ap`: add, selecting parts individually, interactively.
- `br`: branch (obvious)
- `ci`: commit (check in), with `-v` option for clarity
- `cip`: commit, selecting parts individually, interactively.
- `cl`: clone, init submodules (submodules are an advanced topic)
- `di`: diff (obvious)
- `dic`: diff of staging area vs last commit (what is about to be committed)
- `diw`: a word diff, color. Useful for small changes.
- `dis`: a “diffstat”: what files are changed, not contents
- `fe`: fetch (obvious)
- `graph`: show whole git graph (so useful, some of us call it `l`)
- `rem`: remote (obvious)
- `st`: status (obvious)
- `su`: submodule update (advanced)

A useful setting for the `p` aliases:

```
$ git config --global interactive.singlekey true
```

## Advanced aliases

These are advanced aliases and configuration options. We won't explain them, but if you are bored, have some time, or want to go deeper, try to figure out what they do. You might want to check the Git manual pages!

```
$ git config --global alias.cif "commit -v -p --fixup"
$ git config --global alias.rb "rebase --autosquash"
$ git config --global alias.rbi "rebase --interactive --autosquash"
$ git config --global alias.rbis "rebase --interactive --autosquash --autostash"
$ git config --global alias.rbs "rebase --autosquash --autostash"
$ git config --global alias.rec "!git --no-pager log --oneline --graph --decorate
@{upstream}^^^.HEAD"
$ git config --global alias.ls-ignored "ls-files -o -i --exclude-standard"
$ git config --global alias.new "log HEAD..HEAD@{upstream}"
$ git config --global alias.news "log --stat HEAD..HEAD@{upstream}"
$ git config --global alias.newd "log --patch --color-words HEAD..HEAD@{upstream}"
$ git config --global alias.newdi "diff --color-words HEAD...HEAD@{upstream}"
$ git config --global alias.rec "!git --no-pager log --oneline --graph --decorate
@{upstream}^^^.HEAD"
$ git config --global alias.reca "!git --no-pager log --oneline --graph --decorate -n10
--all"
$ git config --global alias.recd "log --decorate --patch @{upstream}^^^.HEAD"
$ git config --global alias.recs "!git --no-pager log --oneline --graph --decorate
@{upstream}^^^.HEAD --stat"
```

## Advanced Git configuration

Besides aliases, you can do plenty of other configuration of `git`. Here are some of the most common ones:

```
$ git config --global interactive.singlekey true
$ git config --global core.pager "less -RS"
$ git config --global core.excludesfile ~/.gitignore
$ git config --global merge.conflictstyle diff3
$ git config --global diff.wordRegex "[a-zA-Z0-9_]+|^[[:space:]]"
$ git config --global diff.mnemonicPrefix true
```

Do you get tired of typing and copying and pasting your remote names all the time, like `git@github.com:myusername`? You can create remote aliases like this:

```
$ git config --global url.git@github.com:.insteadOf gh:
$ git config --global url.git@github.com:/username/.insteadOf ghu:
```

Then, when you add a remote `ghu:recipe`, it will automatically be translated to `git@github.com:/username/recipe` using a simple prefix matching.

## 📌 Keypoints

- If you are frustrated about remembering a command, you should create an alias.

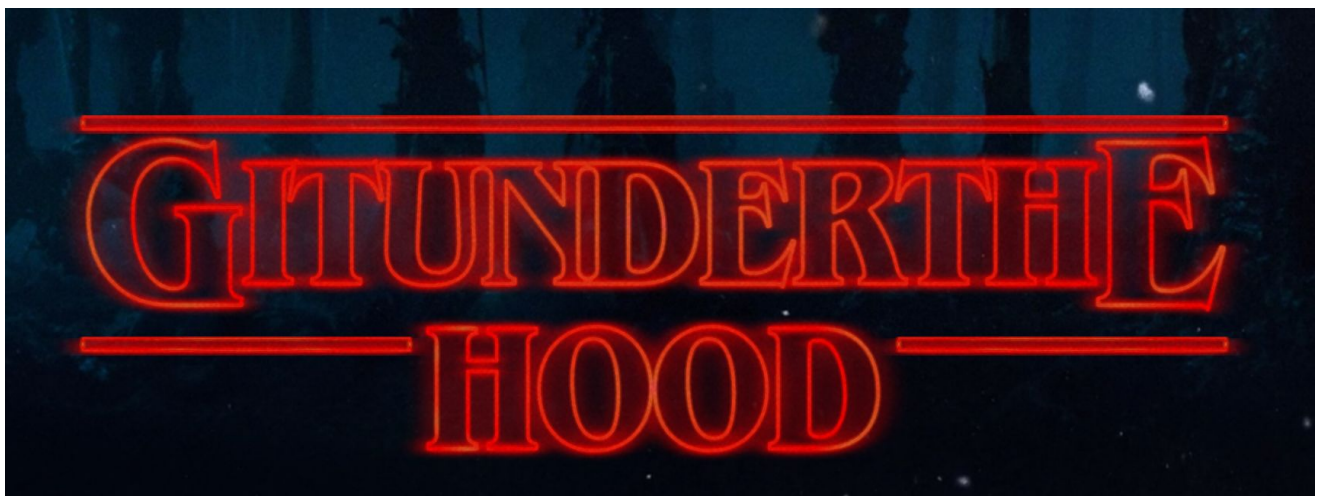
# Git under the hood

## 📌 Objectives

- Verify that branches are pointers to commits and extremely lightweight.

## Instructor note

- 10 min teaching/type-along
- 15 min exercise



## Down the rabbit hole

When working with Git, **you will never need to go inside .git**, but in this exercise we will, in order to learn about how branches are implemented in Git.

For this exercise create a new repository and commit a couple of changes.

Now that we've made a couple of commits let us look at what is happening under the hood.

```

$ cd .git
$ ls -l

drwxr-xr-x  - user 25 Aug 15:51 branches
-rw-r--r-- 499 user 25 Aug 15:52 COMMIT_EDITMSG
-rw-r--r--  92 user 25 Aug 15:51 config
-rw-r--r--  73 user 25 Aug 15:51 description
-rw-r--r--  21 user 25 Aug 15:51 HEAD
drwxr-xr-x  - user 25 Aug 15:51 hooks
-rw-r--r-- 137 user 25 Aug 15:52 index
drwxr-xr-x  - user 25 Aug 15:51 info
drwxr-xr-x  - user 25 Aug 15:52 logs
drwxr-xr-x  - user 25 Aug 15:52 objects
drwxr-xr-x  - user 25 Aug 15:51 refs

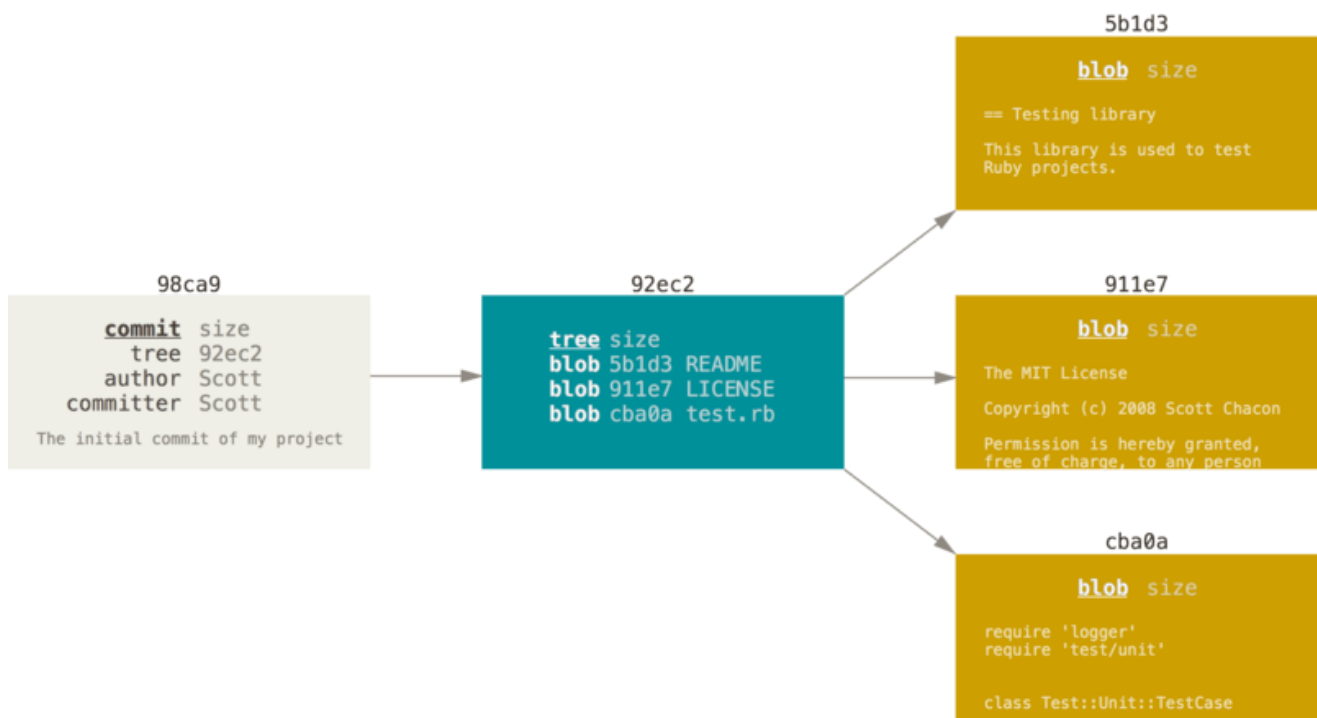
```

Git stores everything under the .git folder in your repository. In fact, **the .git directory is the Git repository**.

Previously when you wrote the commit messages using your text editor, they were in fact saved to `COMMIT_EDITMSG`.

Each commit in Git is stored as a “blob”. This blob contains information about the author and the commit message. The blob references another blob that lists the files present in the directory at the time and references blobs that record the state of each file.

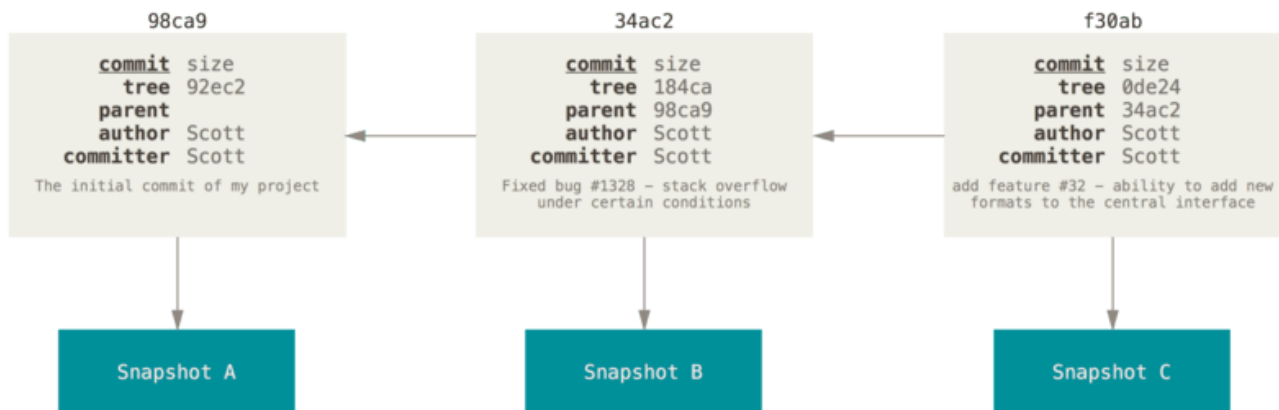
Commits are referenced by a SHA-1 hash (a 40-character hexadecimal string).



States of a Git file. Image from the [Pro Git book](#). License CC BY 3.0.

Once you have several commits, each commit blob also links to the hash of the previous commit. The commits form a **directed acyclic graph** (do not worry if the term is not familiar).





A commit and its parents. Image from the [Pro Git book](#). License CC BY 3.0.

All branches and tags in Git are pointers to commits.

## Git is basically a content-addressed storage system

- CAS: “mechanism for storing information that can be retrieved based on its content, not its storage location”
- Content address is the content digest (SHA-1 checksum)
- Stored data does not change - so when we modify commits, we always create new commits. Git doesn't delete these right away, which is why it is *very hard to lose data if you commit it once*.

Let us poke a bit into raw objects! Start with:

```
$ git cat-file -p HEAD
```

Then explore the `tree` object, then the `file` object, etc. recursively using the hashes you see.

## Demonstration: experimenting with branches

Let us lift the hood and create few branches manually. The goal of this exercise is to hopefully create an “Aha!” moment and provide us a good understanding of the underlying model.

We are starting from the `main` branch and create an `idea` branch:

```
$ git status
```

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

```
$ git switch --create idea
```

```
Switched to a new branch 'idea'
```

```
$ git branch
```

```
* idea  
main
```

Now let us go in:

```
$ cd .git
```

```
$ cd refs/heads
```

```
$ ls -l
```

```
.rw-r--r-- 41 user 25 Aug 15:54 idea
```

```
.rw-r--r-- 41 user 25 Aug 15:52 main
```

Let us check what the `idea` file looks like (do not worry if the hash is different):

```
$ cat idea
```

```
045e3db14740c60684d745e5fb891ae71e335611
```

Now let us replicate this file:

```
$ cp idea idea-2
```

```
$ cp idea idea-3
```

```
$ cp idea idea-4
```

```
$ cp idea idea-5
```

Let us go up two levels and inspect the file `HEAD`:

```
$ cd ../../
```

```
$ cat HEAD
```

```
ref: refs/heads/idea
```

Let us open this file and change it to:

```
ref: refs/heads/idea-3
```

Now we are ready for the **aha moment!** First let us go back to the working area:

```
$ cd ..
```

Now - on which branch are we?

```
$ git branch
```

```
idea
idea-2
* idea-3
idea-4
idea-5
main
```

## Discussion

Discuss the findings with other course participants.

## Quick reference

### Other cheatsheets

- [Detailed 2-page Git cheatsheet](#)
- [Interactive Git cheatsheet](#)

## Glossary

- **working directory/ workspace:** the actual files you see and edit
- **staging area:** Place files go after `git add` and before `git commit`
- **hash:** unique reference of any commit or state
- **branch:** One line of work. Different branches can exist at the same time and split/merge.
- **HEAD:** Pointer to the most recent commit on the current branch.
- **remote:** Roughly, another server that holds .git.
- **origin:** Default name for a remote repository.
- **master:** Default name for main branch on Git. Depending on the configuration and service, the default branch is sometimes **main**. In this lesson we configure Git so that the default branch is called **main** to be more consistent with GitHub and GitLab.

- **main**: Default name for main branch on GitLab and GitHub. In this lesson we configure Git so that the default branch is called **main** to be more consistent with GitHub and GitLab.

## Commands we use

Setup:

- `git config` : edit configuration options
- `git init -b main` : create new repository with `main` as the default branch

See our status:

- `git status` : see status of files - use often!
- `git log` : see history of commits and their messages, newest first
- `git graph` : see a detailed graph of commits. Create this command with `git config --global alias.graph "log --all --graph --decorate --oneline"`
- `git diff` : show difference between working directory and last commit
- `git diff --staged` : show difference between staging area and last commit
- `git show COMMIT` : inspect individual commits

General work:

- `git add FILE` :
  - Add a new file
  - Add a file to staging
- `git commit` : record a version, add it to current branch
- `git commit --amend` : amend our last commit
- `git branch` : show which branch we're on
- `git branch NAME` : create a new branch called "name"
- `git restore FILE` : restore last committed/staged version of FILE, losing unstaged changes
- `git switch --create BRANCH-NAME` : create a new branch and switch to it
- `git revert HASH` : create a new commit which reverts commit HASH
- `git reset --soft HASH` : remove all commits after HASH, but keep their modifications as staged changes
- `git reset --hard HASH` : remove all commits after HASH, permanently throwing away their changes
- `git merge BRANCH-NAME` : merge branch BRANCH-NAME into current branch
- `git grep PATTERN` : search for patterns in tracked files
- `git annotate FILE` : find out when a specific line got introduced and by whom
- `git bisect` : find a commit which broke some functionality

# Customizing Git

## Shell prompt

### Instructor note

Here the instructor can demonstrate how a context-aware and Git-aware shell prompt can look like.

You can make your shell display contextual information about your Git state even at all times.

Here are few example projects that make this possible and easy:

- <https://github.com/jimeh/git-aware-prompt> (bash)
- <https://ohmyz.sh/> (zsh)
- <https://github.com/oh-my-fish/oh-my-fish> (fish)
- <https://github.com/magicmonty/bash-git-prompt> (bash and fish)

## More useful “diff” output

[Delta](#) is a syntax-highlighting pager for git, diff, and grep output. You can customize how you want to highlight the “diff” output. It allows side-by-side view, word-level diff highlighting, improved merge conflict display, and much more.

## Other resources

- [Learn Git branching](#)
- [The entire Pro Git book, written by Scott Chacon and Ben Straub](#)
- [Learn git one commit at a time](#)
- [A successful Git branching model](#)
- [Commit Often, Perfect Later, Publish Once: Git Best Practices](#)
- [PeepCode Git Internals](#)
- [Git Workflows for Pros: A Good Git Guide](#)
- [Branch-per-Feature](#)
- [Git on XKCD](#)
- [An efficient GIT workflow for mid/long term projects](#)

## List of exercises

## Summary

Basics:

- [Exercise: Record changes](#)
- [Optional exercises: Comparing changes](#)

Branching and merging:

- [Exercise: Create and commit to branches](#)
- [Exercise: Merging branches](#)
- [Optional exercises with branches](#)

Conflict resolution:

- [Exercise: Create and resolve a conflict](#)
- [Optional exercises with conflict resolution](#)

Inspecting history:

- [Exercise: Basic archaeology commands](#)
- [Optional exercise: Git bisect](#)

Using the Git staging area:

- [Exercise: Interactive commits](#)
- [Exercise: Using the staging area](#)

Undoing and recovering:

- [Exercise: Revert a commit](#)
- [Exercise: Modify a previous commit](#)
- [Exercise: Git reset](#)

Interrupted work:

- [Exercise: Stashing](#)

## Full list

This is a list of all exercises and solutions in this lesson, mainly as a reference for helpers and instructors. This list is automatically generated from all of the other pages in the lesson. Any single teaching event will probably cover only a subset of these, depending on their interests.

## Instructor guide

### Schedule Day 1

Times here are in CE(S)T.

- 08:50 - 09:00 (10 min) Soft start and icebreaker question
- 09:00 - 09:20 (20 min) Welcome and practical information
- 09:20 - 09:35 (15 min) [Motivation](#)
- 09:35 - 09:50 (15 min) [Basics - configuration and first commits](#)
- 09:50 - 10:00 (10 min) Break
- **10:00 - 10:20 (20 min) Exercise**
  - Record changes
  - Optional exercises
- 10:20 - 10:40 (20 min) [Basics - history, commit log, ignoring](#)
- **10:40 - 11:00 (20 min) Exercise**
  - One or all optional exercises
- 11:00 - 12:00: Lunch break
- 12:00 - 12:15 (15 min) [Branching and merging](#)
- **12:15 - 12:35 (20 min) Exercise**
  - Branch-1
  - Branch-2
- 12:35 - 12:50 (15 min) Summarize/discuss branching and merging
- 12:50 - 13:00 (10 min) Break
- 13:00 - 13:20 (20 min) [Conflict resolution](#)
- 13:20 - 13:30 (10 min) Q&A, feedback, and what will we be doing tomorrow?

## Schedule Day 2

Times here are in CE(S)T.

- 08:50 - 09:00 (10 min) Soft start and icebreaker question
- 09:00 - 09:10 (10 min) Recap and Q&A from day 1
- 09:10 - 09:25 (15 min) [Sharing repositories online](#)
- **09:25 - 09:50 (25 min) Exercise**
  - Set up SSH keys
  - Pushing our guacamole recipe repository to GitHub
  - Clone repository
- 09:50 - 10:00 (10 min) Break
- 10:00 - 10:15 (15 min) [Inspecting history](#)
- **10:15 - 10:45 (30 min) Exercise**
  - History-1
  - History-2 is optional
- 10:45 - 11:00 (15 min) Summarize/discuss inspecting history
- 11:00 - 12:00: Lunch break
- 12:00 - 12:15 (15 min) [Undoing and recovering](#)
- **12:15 - 12:40 (25 min) Exercises**
  - Undoing-1
  - Undoing-2
  - Undoing-3
- 12:40 - 11:50 (10 min) Summarize undoing and recovering and discussion and mention that staging area exists

- 12:50 - 13:00 (10 min) Break
- 13:00 - 13:20 (20 min) [How much Git is necessary?](#)
- 13:20 - 13:30 (10 min) Q&A, feedback, and what will we be doing tomorrow?

## Installation reminders for each day

- Day 1: Git configuration
- Day 2: SSH set up (if you don't, sharing repositories online will be demo)

## Why we teach this lesson

Everyone should be using a version control system for their work, even if they're working alone. There are many version control systems out there, but Git is an industry standard and even if one uses another system chances are high one still encounters Git repositories.

Specific motivations:

- Code easily becomes a disaster without version control
- Mistakes happen - Git offers roll-back functionality and easy backup mechanism
- One often needs to work on multiple things in parallel - branches solve that problem
- Git enables people to collaborate on code or text without stepping on each other's toes
- Reproducibility: You can specify exact versions in publications enabling others to reproduce your work, and if bugs are found one can find out exactly when it was introduced

Many learners in a CodeRefinery workshop have developed code for a few years. A majority have already encountered Git and have used it to some extent, but in most cases they do not yet feel comfortable with it. They lack a good mental model of how Git operates and are afraid of making mistakes. Other learners have never used Git before. This lesson teaches how things are done in Git, which is useful for the newcomers, but also how Git operates (e.g. what commits and branches really are) and what are some good practices (e.g. how to use the staging area), which is useful for more experienced users.

## Intended learning outcomes

By the end of this lesson, learners should:

- realize that version control is very important and Git is a valuable tool to learn and use
- understand that Git is configurable and know how to set basic configurations
- be able to set up Git repositories and make commits
- understand that information on commits, branches etc. are stored under `.git/`, and have a mental model of how that relates to the working directory
- have a mental model of the different states a file can have in Git (untracked, modified, staged, unmodified)
- know how to write good commit messages
- have an idea of how the staging area can be used to craft good commits



- know how to undo commits using `git revert` and discard changes using `git restore`
- understand that `git restore` can be dangerous if changes have not been staged
- know how to create branches and switch between branches
- have a mental model of how branches work and get used to thinking of branches in a graphical (tree-structure) way
- know how to merge branches and understand what that means in terms of combining different modifications
- know how to resolve conflicts, or to abort conflicting merges
- realize that conflicts are generally a good thing since they prevent incorrect merges
- be able to set up a repository on GitHub and connect it with local repository
- push changes to a remote repository
- know a few ways to search through a repository and its history

## How to teach this lesson

### Take first editor steps slowly

Some participants will be new to using a terminal text editor so please open, edit, and close the editor (Nano) slowly in first type-along sessions and exercises to avoid that participants will fall behind the instructor. At one point a student did not follow the file edits of the instructor, and to correct the mistake they had to do a manual merge, which they were not ready for.

### How to use the exercises

Most episodes have standard exercises followed by optional (often more advanced) exercises for more experienced learners so they don't get bored waiting for the newcomers. The instructor should briefly introduce the exercises and mention that after finishing the standard exercise (and indicating that using the green sticky) the learners can move on to the optional ones if they wish. When at least half of the learners have raised the green sticky the instructor should go through the standard exercise to describe its most important take-home messages. It's also fine to briefly go through important points from the optional exercises, but don't spend too much time on it since everyone will not have attempted them.

### “Test your understanding” exercises

Some episodes have a “test your understanding” exercise at the end which is intended as *formative assessment*, i.e. an activity that provides feedback to instructors and learners on whether learning objectives are being met. The instructor should end each episode by posing the “test your understanding” multiple-choice question, giving learners a minute to think about it, and then asking for the right answer or asking learners to raise their hands to signal which answer they think is correct.

### Inspecting history

Key lesson is *how to find when something is broken or what commit has broken the code*.

It can be useful to emphasize that it can be really valuable to be able to search through the history of a project efficiently to find when bugs were introduced or when something changed and why. Also show that `git annotate` and `git show` are available on GitHub and GitLab.

When discussing `git annotate` and `git bisect` the “when” is more important than “who”. It is not to blame anybody but rather to find out whether published results are affected.

Discuss how one would find out this information without version control.

### Questions to involve participants:

- Have you ever found a bug in your code and wondered whether it has affected published results?
- Have you ever wondered when, and by whom, a particular line of code was introduced?
- Have you ever found out that a code behaves differently than it used to but you are not sure when precisely this changed?

### Confusion during `git bisect` exercise:

Learners may get stuck in the `git bisect` exercise if they incorrectly assign a commit as *bad* or *good*. To leave the bisect mode and return to the commit before `git bisect start` was issued, one can do

```
$ git bisect reset
```

and start over if needed.

### Live better than reading the website material

It is better to demonstrate the commands live and type-along. Ideally connecting to examples discussed earlier.

### Log your history in a separate window

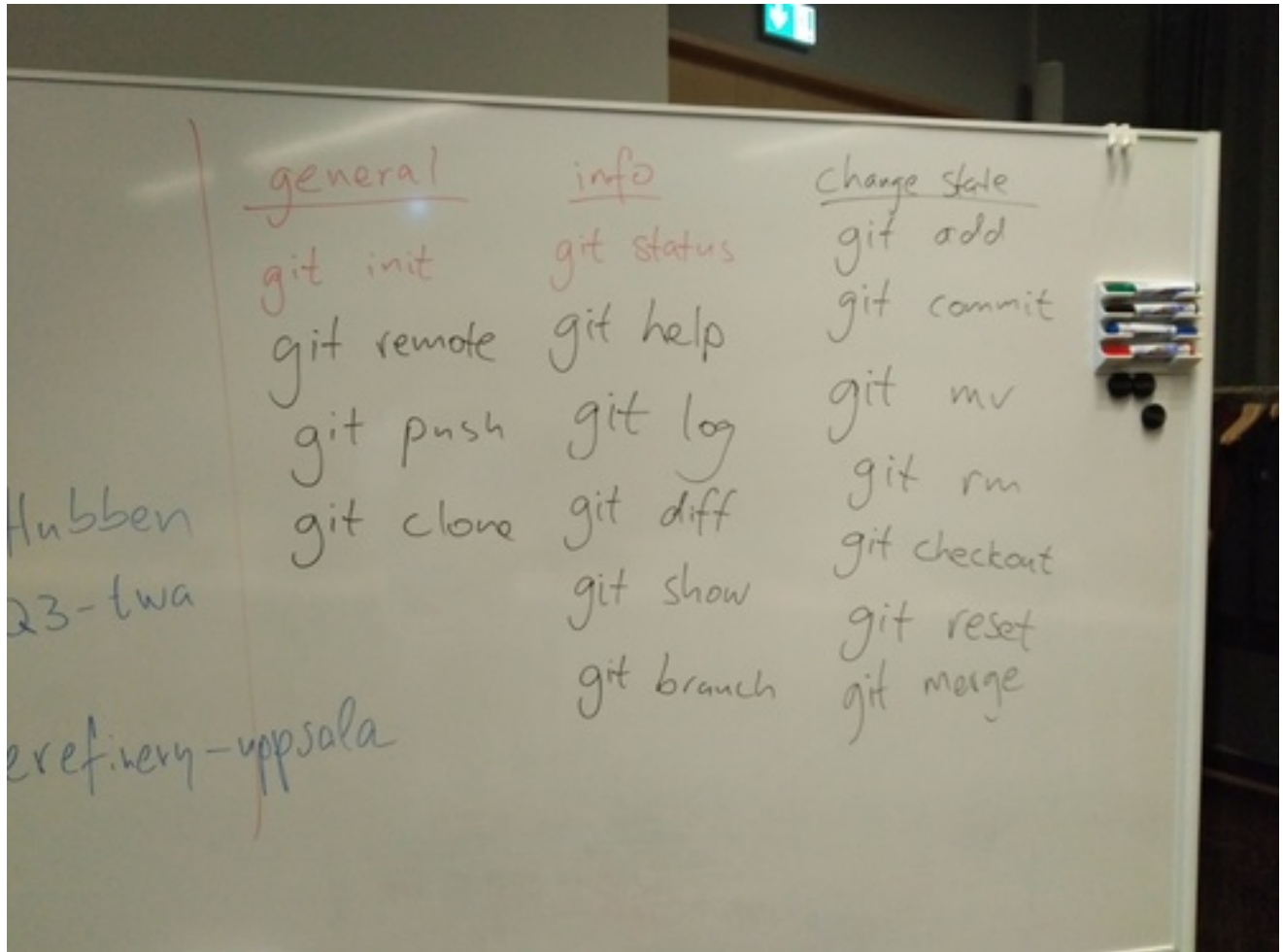
The screencasting (shell window cheatsheet) hints have been moved to the [presenting manual](#).

### Create a cheatsheet on the board

For in-person workshops, create a “cheatsheet” on the board as you go. After each command is introduced, write it on the board. After each module, make sure you haven’t forgotten anything. Re-create and expand in future git lessons. One strategy is:

- a common section for basic commands: `init`, `config`, `clone`, `help`, `stash`
  - info commands, can be run anytime: `status`, `log`, `diff`, `graph`
  - A section for all the commands that move code from different states: `add`, `commit`, etc.
- See the visual cheat sheet below.

You can get inspired by <http://www.ndpsoftware.com/git-cheatsheet.html> to make your cheat sheet, but if you show this make it clear there are far, far more commands on there than you need to know right now., and it's probably too confusing to use after this course. But, the idea of commands moving from the "working dir", "staging area", "commits", etc is good.



Example cheat sheet.

We also recommend to draw simple diagrams up on the board (e.g.: working directory - staging area - repository with commands to move between) and keep them there for students to refer to.

### Draw a graph on the board

Draw the standard commit graphs on the board early on - you know, the thing in all the diagrams. Keep it updated all the time. After the first few samples, you can basically keep using the same graph the whole lesson. When you are introducing new commands, explain and update the graph first, then run `git graph`, then do the command, then look at `git graph` again.

## Repeat the following points

- Always check `git status`, `git diff`, and `git graph` (our alias) before and after every command until you get used to things. These give you a clear view into what is going on, the key to knowing what you are doing. Even after you are used to things... anytime you do something you do infrequently, it's good to check.
- `git graph` is a direct representation of what we are drawing on the board and should constantly be compared to it.
- Once you `git add` something, it's almost impossible to lose it. This is used all the time, for example once you commit or even add it is hard to lose. Commit before you merge or rebase. And so on.

## Start from identical environment

You probably have a highly optimized bash and git environment - one that is different from students. Move `.gitconfig` and `.bashrc` out of the way before you start so that your environment is identical to what students have.

### ❗ Why GitHub?

In this introduction we will mention and use [GitHub](#) but also [GitLab](#) and [Bitbucket](#) allow similar workflows and basically everything that we will discuss is transferable. With this material and these exercises we do not implicitly endorse the company [GitHub](#). We have chosen to demonstrate a number of concepts using examples with [GitHub](#) because it is currently the most popular web platform for hosting Git repositories and the chance is high that you will interact with [GitHub](#)-based repositories even if you choose to host your Git repository on another platform.