Update README.md
gohr authored just now

5504005c

README.md 77.4 KB

Git 101: For Absolute Beginners

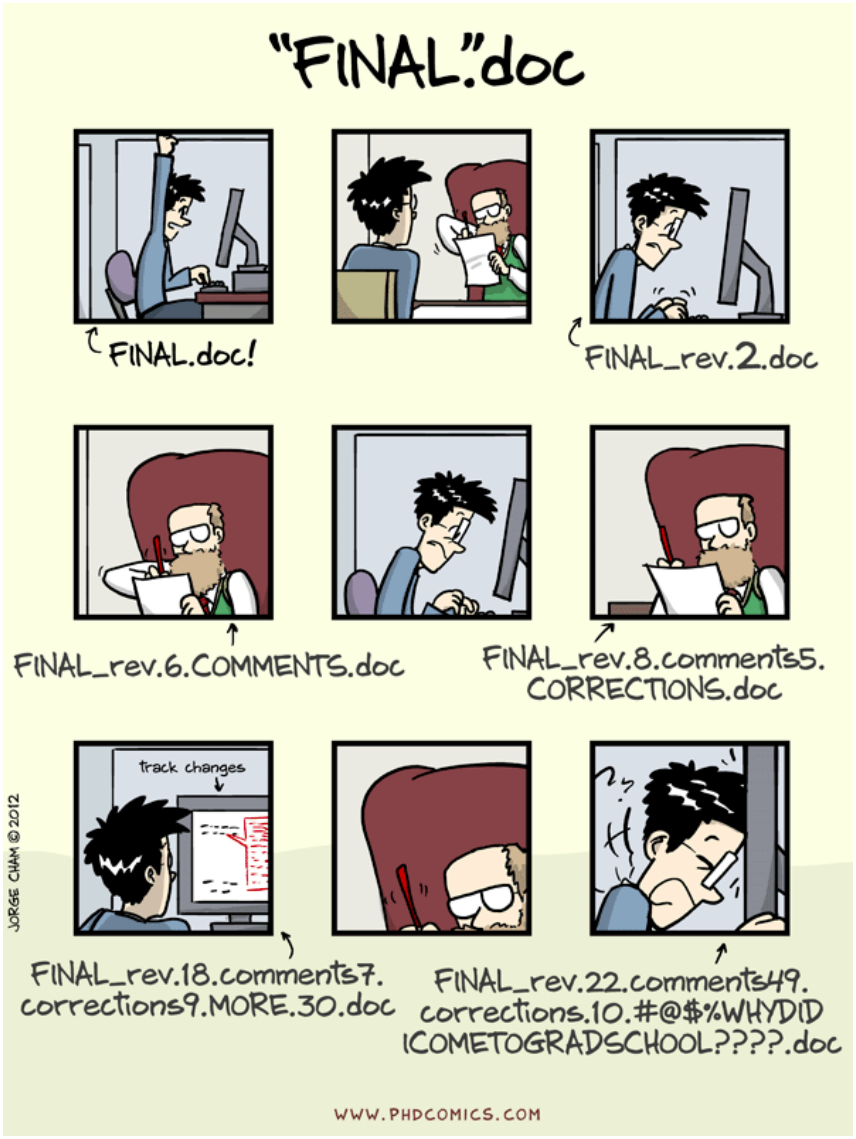
- Offered by the Scientific Computing Facility @ CBG.
- Contact: `scicomp@mpi-cbg.de`

- **Learning objectives**
  - Concept behind working with Git version control
  - Use Git version control for local projects/files
  - Make snapshots of files while project evolves
  - Check log messages from last commits
  - Checkout previous versions of files
  - Investigate differences between versions of files
  - Remote repositories on Git server like GitHub or GitLab
  - Connect local Git repository with remote Git repository
  - Clone Git repository from GitHub or GitLab locally
  - Work with remote Git repositories
  - Collaborate with colleagues on one project
  - Resolve conflicts
- **For whom and which prerequisites**
  - CBG/CSBD affiliated
  - Students should feel comfortable with basic work in the terminal, eg. course Bash 101 for beginners
- **Course format**
  - Self-study
    - Students read material and do exercises themselves
    - In case of questions or wish to discuss particular points, write an Email to schedule a Zoom chat with us: `scicomp@mpi-cbg.de`
  - Online or class-room course
    - Offered on demand
    - Minimum number of participants: 5
    - Write us an Email to evince your interest: `scicomp@mpi-cbg.de`
- **Installation before class on your computer**
  - Mac/Linux users:
    - `git` must be installed
      - to check run in terminal: `git version`
      - installation instruction
  - Windows users
    - Windows Subsystem for Linux (WSL) Details
      1. open an admin CMD
      2. wsl --install -d ubuntu
      3. wcl --set-version-default 2
    - `git` should be installed already with WSL
      - to check run in WSL terminal: `git version`
      - follow installation instructions for Linux inside your WSL
- **Origin of material**

- Taken and partly adapted from the software carpentries.

Part 1: Git Basics for One-User Local Use

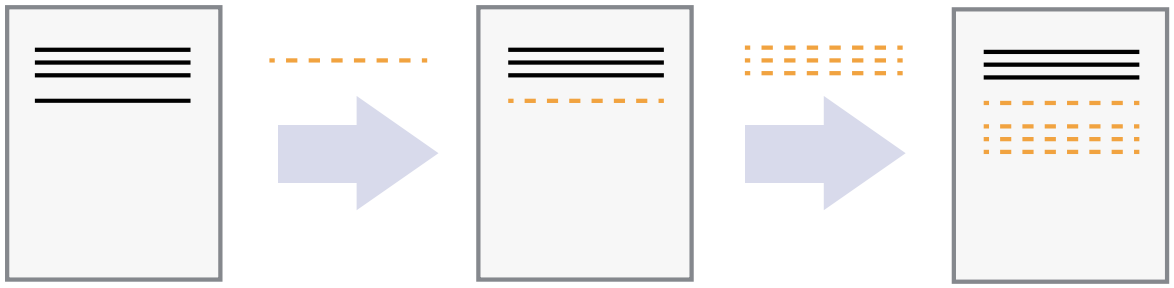
Working on projects or "Why Git?"



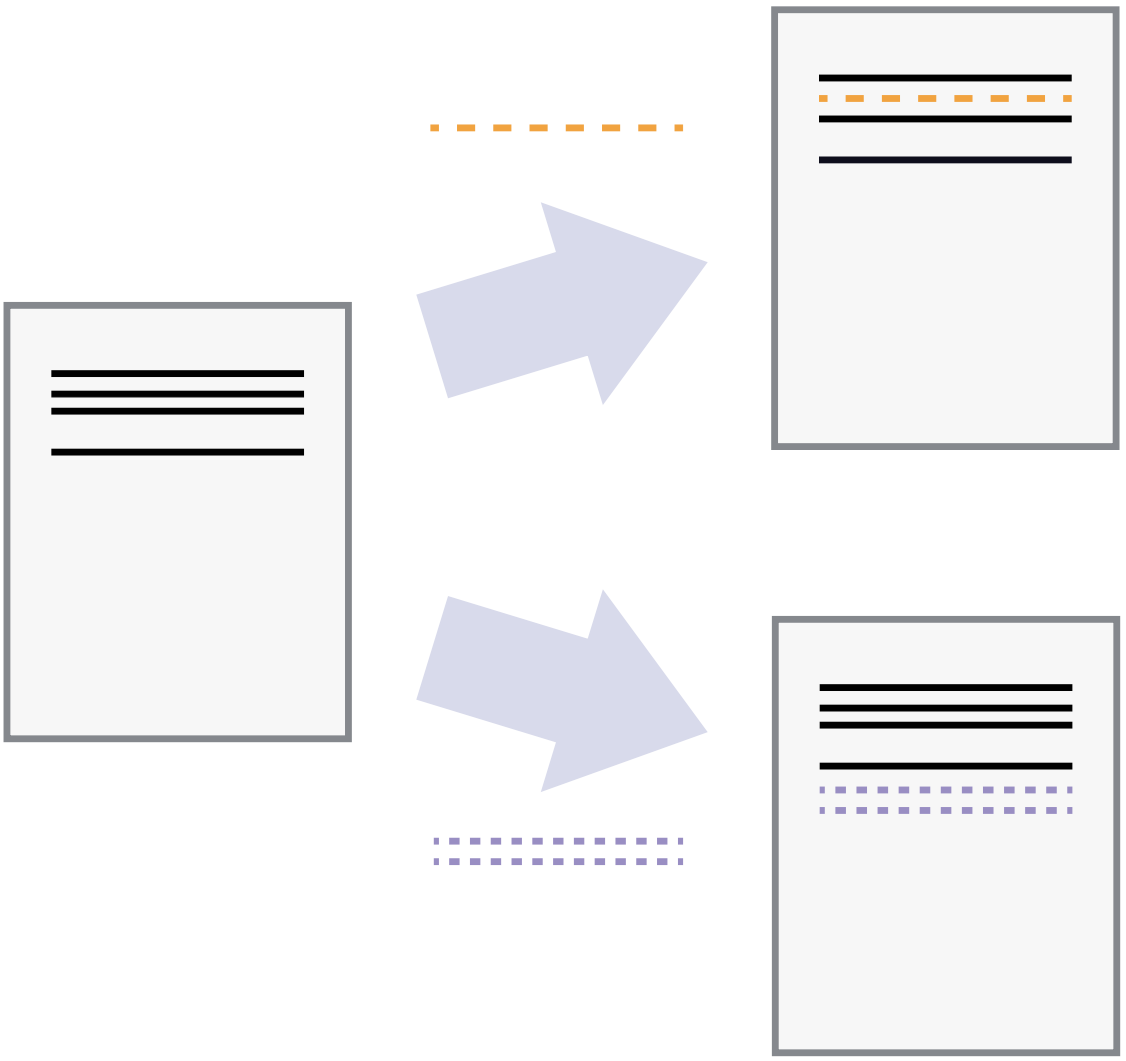
We've all been in this situation before: it seems unnecessary to have multiple nearly-identical versions of the same document. Some word processors let us deal with this a little better, such as Microsoft Word's Track Changes, Google Docs' version history, or LibreOffice's Recording and Displaying Changes.

What we want	What we do	Problem	What we need
Keep previous versions	Keep copies of entire documents	Management overhead with thousands of files	Be able to manage versions in a neat way
Annotate previous versions	Put annotation into file names	Long, hard te deal with file names	Have build-in function for version annotation
Collaborate efficiently	2-3 people: sequential work more: GoogleDocs	Slow, hard to keep track who contributed which change when and why	System for parallel contributions that allows us to manage all changes

Version control systems start with a base version of the document and then record changes you make each step of the way. You can think of it as a recording of your progress: you can rewind to start at the base document and play back each change you made, eventually arriving at your more recent version.

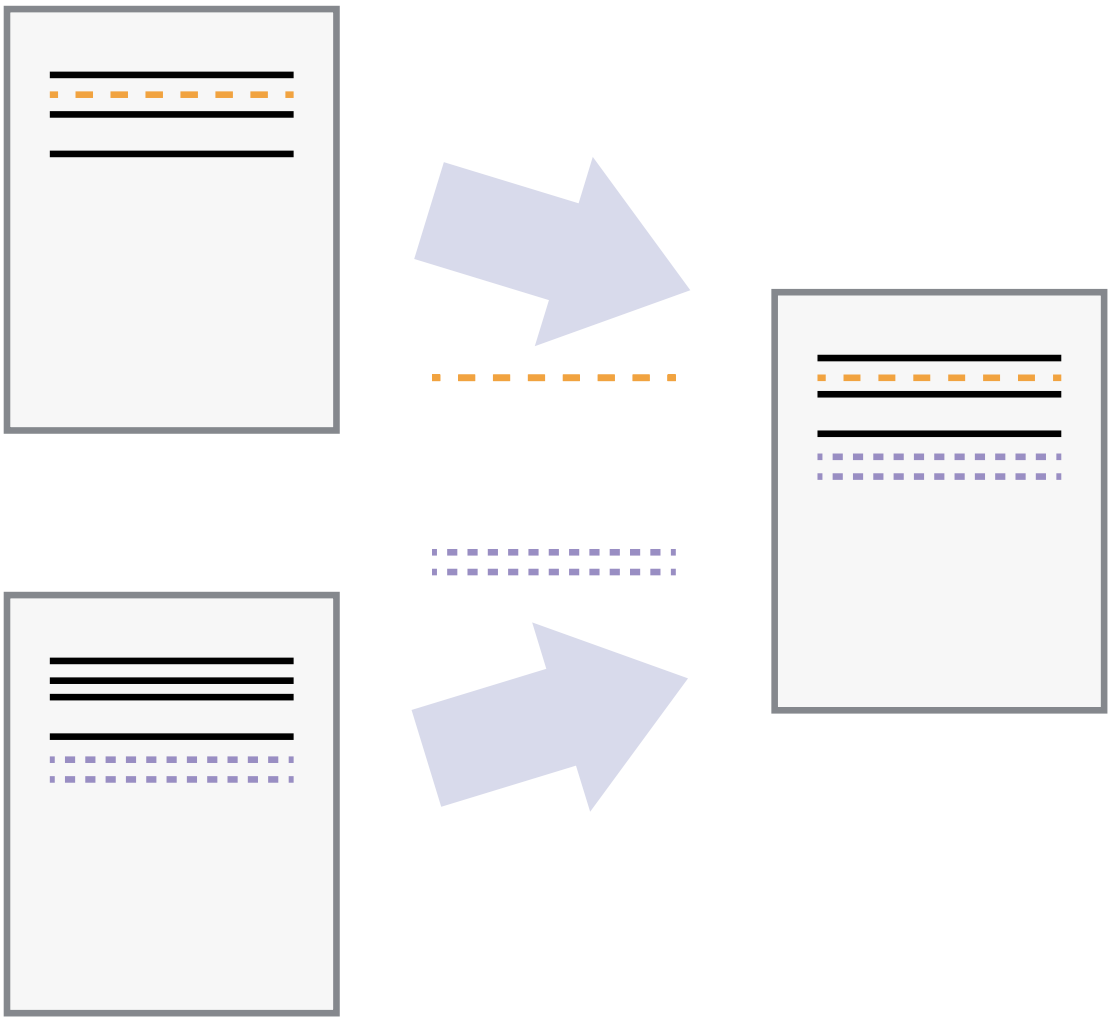


Once you think of changes as separate from the document itself, you can then think about “playing back” different sets of changes on the base document, ultimately resulting in different versions of that document. For example, two users can make independent sets of changes on the same document.



Unless multiple users make changes to the same section of the document - a conflict - you can incorporate two sets

of changes into the same base document.

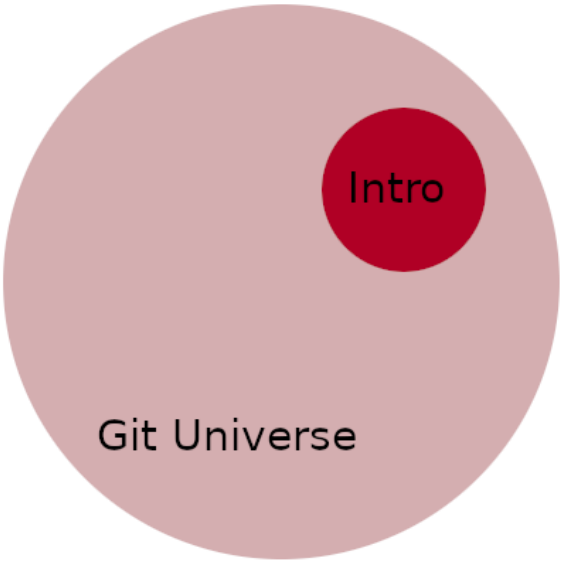


A version control system is a tool that keeps track of these changes for us, effectively creating different versions of our files. It allows us to decide which changes will be made to the next version (each record of these changes is called a commit), and keeps useful metadata about them. The complete history of commits for a particular project and their metadata make up a repository. Repositories can be kept in sync across different computers, facilitating collaboration among different people.

Key points:

- Version control is like an unlimited ‘undo’
- Version control also allows many people to work in parallel
- File and directory names should contain only: A-Za-z0-9_

This introductary course on Git



Key points:

- Provide you with basic understanding and working patterns
- Get you started
- Empower you to learn by yourself as you go on: Learning by doing

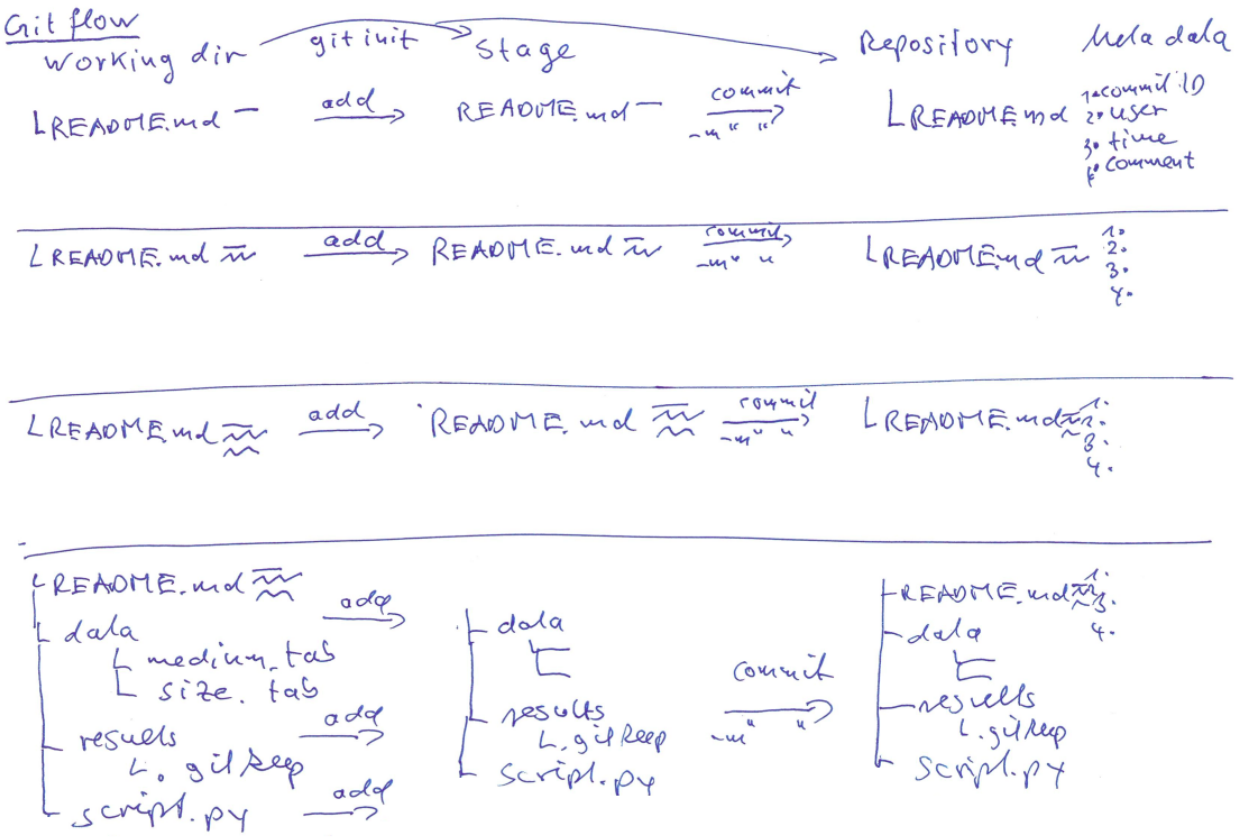
Setting up Git

- Follow instructions on SW Carpentries | Git Novice | Setting up Git
- Important: When you work on **Windows with the Windows Subsytem for Linux**, you should configure Line Endings for MacOS/Linux not for Windows because you are essentially working with a Linux system

Key points:

- Git needs to know who you are because changes you make will be annotated with information about you
- Use git config with the `--global` option to configure a user name, email address, editor, and other preferences once per machine.
- Or with `--local` individually for each repository

The Git flow: status, add, commit



Key points:

- most important Git commands: `git init`, `git status`, `git add`, `git commit`

Creating a Git repository

Create folder to contain all of our exercises and therein a folder for project X where you want to track files of that project. Change into that folder.

```
pwd
mkdir git_course
```

```
cd git_course
mkdir projx
cd projx
```

Initialize this folder `projx` to be under Git version control.

```
git init
```

It is important to note that git init will put under version control folder `projx` and any sub-folder therein. No need to use `git init` in sub-folders again, whether sub-folders are present from the beginning or added later. Also, note that the creation of the folder `projx` and putting it under Git version control are completely separate processes.

If we use `ls` to show the folder's contents, it appears that nothing has changed.

```
ls
```

But if we add the `-a` flag to show everything, we can see that Git has created a hidden directory within planets called `.git`

```
ls -a

# Output
.       ..      .git
```

Git uses this special sub-folder to store all the information about the project including all files and sub-folders located within the project's directory. If we ever delete the .git subdirectory, we will lose the project's history.

We can check that everything is set up correctly by asking Git to tell us the status of our project:

```
git status

# Output
On branch main

No commits yet

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

If you are using a different version of git, the exact wording of the output might be slightly different.

Exercise

Assume you want to track also some small data files that belong the projext X. Read carefully the following sequence of commands and try to answer the question below.

```
cd git_course  # return to folder git_course from where we started
cd projx       # go into folder projx , which is already under Git version control
ls -a          # ensure the .git subdirectory is still present
mkdir data     # make a sub-folder projx/data
cd data        # go into sub-folder data
git init       # put sub-folder data under Git version control
ls -a          # ensure the .git sub-folder is present
```

Is the `git init` command run inside sub-folder `data` required for tracking files stored in sub-folder `data`?

▾ Answer:

No, it is not. Once initiated, users can add all sub-directories and files therein to Git vertion control.

How can you undo the last command `git init` in the sub-folder `data`?

Git repositories can interfere with each other if they are nested: the outer repository will try to version-control the inner repository. Therefore, it's best to create each new Git repository in a separate directory. To be sure that there is no conflicting repository in the directory, check the output of git status. If it looks like the following, you are good to

go to create a new repository as shown above:

```
git status

# Output
fatal: Not a git repository (or any of the parent directories): .git
```

Removing files from a Git repository needs to be done with caution. But we have not learned yet how to tell Git to track a particular file. Files that are not tracked by Git can easily be removed like any other ordinary files with

```
rm file_name

rm -rf folder_name
```

Key points:

- `git init` : put a directory under version control
- Git stores all of its repository data in the .git directory

Tracking files and their changes

First let's make sure we are still in the right folder `git_course/projx` :

```
pwd
cd git_course/projx
```

At the beginning of the project it is a good idea to create file `README.md` which shall contain important information on the project. This file will grow as the project advances. We'll use here `vim` to edit the file. You can use any editor you like. Remember, the bash command to create or edit a new file will depend on the editor you choose (it might not be `vim`).

```
vim README.md
```

Type the text below into the file `README.md` and save that file and leave the editor.

```
# Project X
```

Let's first verify that the file was properly created:

```
ls
```

```
# Output
README.md
```

`README.md` contains a single line which we can see by running:

```
cat README.md

# Output
# Project X
```

If we check the status of our folder `projx` again, Git tells us that it has noticed the new file:

```
git status

# Output
On branch main

No commits yet

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

        README.md

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

The untracked-files message means that there's a file in the folder that Git isn't keeping track of. We can tell Git to track a file using `git add`:

```
git add README.md
```

and then check again the status:

```
git status

# Output
On branch main

No commits yet

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

        new file:   README.md
```

Git now knows that it's supposed to keep track of `README.md`, but it hasn't recorded these changes yet as a commit or snapshot. To do so:

```
git commit -m "Adds README.md"

# Output
[main (root-commit) f22b25e] Adds README.md
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

When we run `git commit`, Git takes everything we have told it to save by using `git add` before and stores a copy permanently inside the special `.git` directory. This permanent copy is called a commit (or revision) and its short identifier is `f22b25e`. Your commit may have another identifier.

We use the `-m` flag (for message) to record a short, descriptive, and specific comment that will help us remember later what we changed and why. If we just run git commit without the `-m` option, Git will launch `nano` (or whatever other editor we configured as core.editor) so that we can write a longer message.

Good commit messages start with a brief (< 50 characters) statement about the changes made in the commit. Generally, the message should complete the sentence "If applied, this commit will ... " . If you want to go into more

detail, add a blank line between the summary line and your additional notes. Use this additional space to explain why you made changes and/or what their impact will be.

If we run `git status` now:

```
git status

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

it tells us everything is up to date. If we want to know what we've done recently, we can ask Git to show us the project's history using `git log`:

```
git log

# Output
commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: First and last name <Email>
Date:   Thu Aug 22 09:51:46 2013 -0400

   Adds README.md
```

`git log` lists all commits made to a repository in reverse chronological order. The listing for each commit includes the commit's full identifier (which starts with the same characters as the short identifier printed by the git commit command earlier), the commit's author, when it was created, and the log message Git was given when the commit was created.

If we run `ls` at this point, we will still see just one file called `README.md`. That's because Git saves information about files' history in the special sub-folder `.git` mentioned earlier so that our filesystem doesn't become cluttered (and so that we can't accidentally edit or delete an old version). There will be never a good reason for going into that sub-folder `.git` and do anything therein. You can ignore it.

Now suppose you want to add more information like the project owner to `README.md`. (Again, we'll edit with `vim` and then cat the file to show its contents; you may use a different editor, and don't need to cat.)

```
vim README.md
cat README.md

# Output
# Project X
- owner: I
```

When we run `git status` now, it tells us that a file it already knows about has been modified:

```
git status

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

        modified:   README.md

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

The last line is the key phrase: "no changes added to commit". We have changed this file, but we haven't told Git we will want to save those changes (which we do with `git add`) nor have we saved them (which we do with `git commit`). So let's do that now. It is good practice to always review our changes before saving them. We do this using `git diff`. This shows us the differences between the current state of the file and the most recently saved version:

```
git diff

# Output
diff --git a/README.md b/README.md
```

```
index df0654a..315bf3a 100644
--- README.md
+++ README.md
@@ -1 +1,2 @@
 # Project X
+- owner: I
```

The output is cryptic because it is actually a series of commands for tools like editors and patch telling them how to reconstruct one file given the other. If we break it down into pieces:

1. The first line tells us that Git is producing output similar to the Unix diff command comparing the old and new versions of the file.
2. The second line tells exactly which versions of the file Git is comparing; df0654a and 315bf3a are unique computer-generated labels for those versions.
3. The third and fourth lines once again show the name of the file being changed.
4. The remaining lines are the most interesting, they show us the actual differences and the lines on which they occur. In particular, the + marker in the first column shows where we added a line.

After reviewing our change, it's time to commit it:

```
git commit -m "Adds project owner to README"

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

        modified:   README.md

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

Git won't commit because we didn't use `git add` first. Let's fix that:

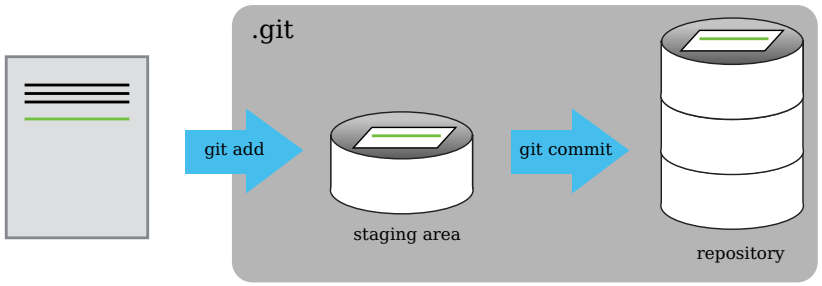
```
git add README.md
git commit -m "Adds project owner to README"

# Output
[main 34961b1] Adds project owner to README
 1 file changed, 1 insertion(+)
```

Git insists that we add files to the set we want to commit before actually committing anything. This allows us to commit our changes in stages and capture changes in logical portions rather than only large batches. For example, suppose we're adding a few citations to relevant research to our thesis. We might want to commit those additions, and the corresponding bibliography entries, but not commit some of our work drafting the conclusion (which we haven't finished yet).

To allow us to do this, Git has a special staging area where it keeps track of things that have been added to the current changeset but not yet committed.

Staging area

If you think of Git as taking snapshots of changes over the life of a project, `git add` specifies what will go in a snapshot (putting things in the staging area), and `git commit` then actually takes the snapshot, and makes a permanent record of it (as a commit). If you don't have anything staged when you type `git commit`, Git will prompt you to use `git commit -a or git commit --all`, which is kind of like gathering everyone to take a group photo! However, it's almost always better to explicitly add things to the staging area, because you might commit changes you forgot you made. (Going back to the group photo simile, you might get an extra with incomplete makeup walking on the stage for the picture because you used `-a`!) Try to stage things manually, or you might find yourself searching for `git undo commit` more than you would like!



Let's watch as our changes to a file move from our editor to the staging area and into long-term storage. First, we'll add another line to the file:

```
vim README.md
cat README.md

# Output
# Project X
- owner: I
- start date: today
```

```
git diff

# Output
diff --git a/README.me b/README.me
index a751688..c72c15a 100644
--- a/README.me
+++ b/README.me
@@ -1,2 +1,3 @@
 # Project X
 - owner: I
+- start date: today
```

So far, so good: we've added one line to the end of the file (shown with a + in the first column). Now let's put that change in the staging area and see what `git diff` reports:

```
git add README.md
git diff
```

There is no output: as far as Git can tell, there's no difference between what it's been asked to save permanently and what's currently in the directory. However, if we do this:

```
git diff --staged

# Output
diff --git a/README.me b/README.me
```

```
index a751688..c72c15a 100644
--- a/README.me
+++ b/README.me
@@ -1,2 +1,3 @@
 # Project X
 - owner: I
+- start date: today
```

it shows us the difference between the last committed change and what's in the staging area. Let's save our changes:

```
git commit -m 'adds start date to README'

# Output
[master 69ace43] adds start date to README
 1 file changed, 1 insertion(+)
```

check our status:

```
git status

# Output
On branch master
nothing to commit, working tree clean
```

and look at the history of what we've done so far:

```
git log

# Output
commit 11bafffafeb2266aa7ee614a38f2c9c4768b2d62 (HEAD -> master)
Author: First and last name <Email>
Date:   Tue Jul 12 16:41:56 2022 +0200

    adds start date to README

commit 9cef64f9dcbab02b4101477e4938624e06cc21e4
Author: First and last name <Email>
Date:   Tue Jul 12 16:41:37 2022 +0200

    adds owner to README

commit f58ecfb32afe4a38a6b2de40535bdbabdfc0c51b
Author: First and last name <Email>
Date:   Tue Jul 12 16:41:17 2022 +0200

    adds README.md
```

Word-based diffing

Sometimes, e.g. in the case of the text documents a line-wise diff is too coarse. That is where the `--color-words` option of `git diff` comes in very useful as it highlights the changed words using colors.

Limit log size

To avoid having `git log` cover your entire terminal screen, you can limit the number of commits that Git lists by using `-N`, where `N` is the number of commits that you want to view. For example, if you only want information from the last commit you can use:

```
git log -1

# Output
Author: First and last name <Email>
Date:   Tue Jul 12 16:41:56 2022 +0200

    adds start date to README
```

You can also reduce the quantity of information using the `--oneline` option:

```
git log --oneline

# Output
11bafff (HEAD -> master) adds start date to README
9cef64f adds owner to README
f58ecfb adds README.md
```

You can also combine the `--oneline` option with others. One useful combination adds `--graph` to display the commit history as a text-based graph and to indicate which commits are associated with the current HEAD, the current branch main, or other Git references:

```
git log --oneline --graph

# Output
* 11bafff (HEAD -> master) adds start date to README
* 9cef64f adds owner to README
* f58ecfb adds README.md
```

Folders

Two important facts you should know about folders in Git.

1. Git does not track folders on their own, only files within them. Try it for yourself:

```
mkdir results
git status
git add results
git status
```

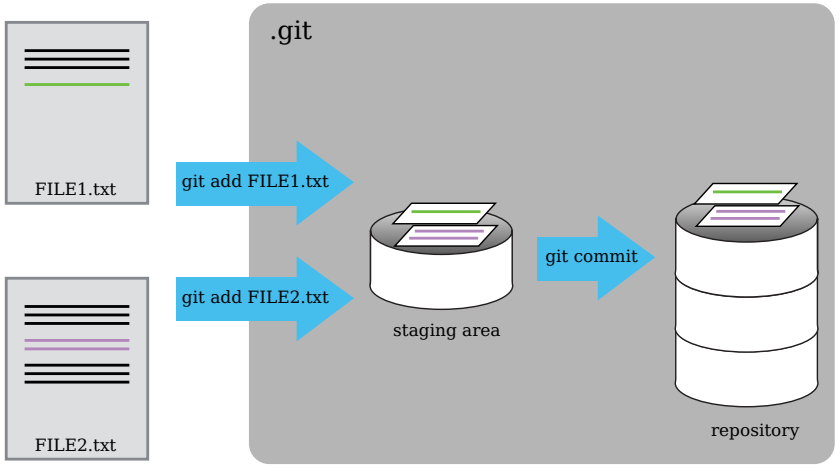
Note, our newly created empty folder `results` does not appear in the list of untracked files even if we explicitly add it (via `git add`) to our repository. This is the reason why you will sometimes see files `.gitkeep` in otherwise empty folders. Unlike `.gitignore`, these files are not special and their sole purpose is to populate a folder so that Git adds it to the repository. In fact, you can name such files anything you like.

```
touch results/.gitkeep
git status
git add results
git commit -m 'adds folder results'
git status
```

2. If you create a folder in your Git repository and populate it with files, you can add all files in the directory at once by:

```
git add <folder-with-files>
```

To recap, when we want to add changes to our repository, we first need to add the changed files to the staging area (`git add`) and then commit the staged changes to the repository (`git commit`):



Exercise

Which of the following commit messages would be most appropriate for the last commit made to README.md?

1. Changes
2. Added line '-start date: today' to README.md
3. Adds start date to README

▾ Answer:

Answer 3

Exercise

Which command(s) below would save the changes of `myfile.txt` to my 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"
```

▾ Answer:

Answer 3

Commiting multiple files

The staging area can hold changes from any number of files that you want to commit as a single snapshot.

1. Add one more line to README.md, eg. the name of your boss
2. Create a new file `script.py` (Python script) and add as very first line the Python shebang `#!/urs/bin/env python`
3. Create an empty file `LICENSE_draft` where you plan to add the software license under which you want to release your Python analysis script
4. Add changes from all files to the staging area, and commit those changes.

`bio` project

1. Create a new folder `bio` inside `git_course` and put it under Git version control
2. Write a three-line biography for yourself in a file called `me.txt`, commit your changes
3. Modify one line, add a fourth line
4. Display the differences between its updated state and its original state.

Key points:

- `git status` shows the status of a repository
- Files can be stored in a project's working directory (which users see), the staging area (where the next commit is being built up) and the local repository (where commits are permanently recorded)
- `git add` puts files on the staging area
- `git commit` saves the staged content as a new commit in the local repository
- Write a commit message that accurately describes your changes

Exploring the project history and recovering previous versions

As we saw in the previous episode, we can refer to commits by their identifiers. You can refer to the most recent commit of the working directory by using the identifier `HEAD`.

We added one line at a time to `README.md`, so it's easier to track our progress. Before we start, let's make a change to `README.md`, adding yet another line.

```
vim README.md
cat README.md

# Output
# Project X
- owner: I
- start date: today
- a change we will regret later
```

Now, let's see what we get when we compare to the last saved version:

```
git diff HEAD README.md

# Output
diff --git a/projx/README.md b/projx/README.md
index c72c15a..e2e3201 100644
--- a/projx/README.md
+++ b/projx/README.md
@@ -1,3 +1,4 @@
 # Project X
 - owner: I
 - start date: today
+- a change we will regret later
```

which is the same as what you would get if you leave out `HEAD` (try it). The real goodness in all this is how you can refer to previous commits. We do that by adding `~1` (where `~` is tilde, pronounced [til-duh]) to refer to the commit one before `HEAD`.

```
git diff HEAD~1 README.md

# Output
diff --git a/projx/README.md b/projx/README.md
index a751688..e2e3201 100644
--- a/projx/README.md
+++ b/projx/README.md
@@ -1,2 +1,4 @@
 # Project X
 - owner: I
+- start date: today
```

```
+- a change we will regret later
```

If we want to see the differences between older commits we can use git diff again, but with the notation `HEAD~1`, `HEAD~2`, and so on, to refer to them:

```
git diff HEAD~2 README.md

# Output
diff --git a/projx/README.md b/projx/README.md
index a5433fd..e2e3201 100644
--- a/projx/README.md
+++ b/projx/README.md
@@ -1 +1,4 @@
 # Project X
+- owner: I
+- start date: today
+- a change we will regret later
```

We could also use `git show` which shows us what changes we made at an older commit as well as the commit message, rather than the differences between a commit and our working directory that we see by using `git diff`.

```
git show HEAD~2 README.md

# Output
commit f9e8a4063218bd499d833eace629f86269aa6ae6
Author: First and last name <Email>
Date:   Wed Jul 13 10:21:42 2022 +0200

    Adds README

diff --git a/projx/README.md b/projx/README.md
new file mode 100644
index 0000000..a5433fd
--- /dev/null
+++ b/projx/README.md
@@ -0,0 +1 @@
+# Project X
```

In this way, we can build up a chain of commits. The most recent end of the chain is referred to as `HEAD`; we can refer to previous commits using the `~` notation, so `HEAD~1` means “the previous commit”, while `HEAD~123` goes back `123` commits from where we are now.

We can also refer to commits using those long strings of digits and letters that `git log` displays. These are unique IDs for the changes, and unique really does mean unique: every change to any set of files on any computer has a unique 40-character identifier. Our first commit was given the ID f9e8a4063218bd499d833eace629f86269aa6ae6, so let’s try this:

```
git diff f9e8a4063218bd499d833eace629f86269aa6ae6 README.md

# Output
86269aa6ae6 README.md
diff --git a/projx/README.md b/projx/README.md
index a5433fd..e2e3201 100644
--- a/projx/README.md
+++ b/projx/README.md
@@ -1 +1,4 @@
 # Project X
+- owner: I
+- start date: today
+- a change we will regret later
```

That’s the right answer, but typing out random 40-character strings is annoying, so Git lets us use just the first few characters (typically seven for normal size projects):

```
git diff f9e8a40 README.md
```

```
# Output
diff --git a/projx/README.md b/projx/README.md
index a5433fd..e2e3201 100644
--- a/projx/README.md
+++ b/projx/README.md
@@ -1 +1,4 @@
 # Project X
+- owner: I
+- start date: today
+- a change we will regret later
```

All right! So we can save changes to files and see what we've changed. Now, how can we restore older versions of things? Let's suppose we change our mind about the last update to README.txt (the "ill-considered change").

`git status` now tells us that the file has been changed, but those changes haven't been staged:

```
git status

# Output
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

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

We can put things back the way they were by using git checkout:

```
git checkout HEAD README.md
cat README.md

# Output
# Project X
- owner: I
- start date: today
```

As you might guess from its name, `git checkout` checks out (i.e., restores) an old version of a file. In this case, we're telling Git that we want to recover the version of the file recorded in `HEAD`, which is the last saved commit. If we want to go back even further, we can use a commit identifier instead:

```
git checkout f9e8a40632 README.md
cat README.md

# Output
# Project X

git status

# Output
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md
```

Notice that the changes are currently in the staging area. Again, we can put things back the way they were by using `git checkout`:

```
git checkout HEAD README.md
cat README.md

# Output
# Project X
- owner: I
- start date: today
```

Don't lose your HEAD

Above we used:

```
git checkout f9e8a40632 README.md
```

to revert `README.md` to its state after the commit `f9e8a40632`. But be careful! The command checkout has other important functionalities and Git will misunderstand your intentions if you are not accurate with the typing. For example, if you forget `README.md` in the previous command.

```
git checkout f9e8a40632

# Output
Note: switching to 'f9e8a40632'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at f9e8a40 Adds README
```

The `detached HEAD` is like "look, but don't touch", so you shouldn't make any changes in this state. After investigating your repo's past state, reattach your `HEAD` with `git checkout master` or `git checkout main`.

```
git checkout master

# Output
Previous HEAD position was f9e8a40 Adds README
Switched to branch 'master'
```

Simplifying the common case

After checking out the `HEAD` version of `README.md` we lost the last line we added `- a change we will regret later`. Another way to undo changes in files that were not yet saved in the repository is the following:

```
git checkout -- README.md
```

As it says, `git checkout` without a version identifier restores files to the state saved in `HEAD`. The double dash `--` is needed to separate the names of the files being recovered from the command itself: without it, Git would try to use the name of the file as the commit identifier.

Whenever you worked on a file and messed it up but haven't yet put the latest changes to the staging area you can restore the last saved version of that file with `git checkout -- <FILE>` and all of your recent changes are gone.

The fact that files can be reverted one by one tends to change the way people organize their work. If everything is in one large document, it’s hard (but not impossible) to undo changes to the introduction without also undoing changes made later to the conclusion. If the introduction and conclusion are stored in separate files, on the other hand, moving backward and forward in time becomes much easier.

Exercise

Jennifer has made changes to the Python script that she has been working on for weeks, and the modifications she made this morning “broke” the script and it no longer runs. She has spent ~ 1hr trying to fix it, with no luck.

Luckily, she has been keeping track of her project’s versions using Git. Which commands below will let her recover the last committed version of her Python script called script.py?

1. git checkout HEAD
2. git checkout HEAD script.py
3. git checkout HEAD~1 script.py
4. git checkout script.py
5. git checkout -- script.py
6. 2, 4, and 5

▾ Answer:

The answer is (6) 2, 4, and 5

The `checkout` command restores files from the repository, overwriting the files in your working directory. Answers 2, 4 and 5 restore the latest version in the repository of the file script.py. Answer 2 uses `HEAD` to indicate the latest, whereas answer 4 uses the unique ID of the last commit, which is what `HEAD` means.

Answer 3 gets the version of script.py from the commit before `HEAD`, which is NOT what we wanted.

Answer 1 can be dangerous! Without a filename, `git checkout` will restore all files in the current directory (and all directories below it) to their state at the commit specified. This command will restore script.py to the latest commit version, but it will also restore any other files that are changed to that version, erasing any changes you

may have made to those files! As discussed above, you are left in a detached `HEAD` state, and you don't want to be there.

Exercise

Jennifer is collaborating with colleagues on her Python script. She realizes her last commit to the project's repository contained an error, and wants to undo it. Jennifer wants to undo correctly so everyone in the project's repository gets the correct change. The command `git revert <erroneous commit ID>` will create a new commit that reverses the erroneous commit.

The command `git revert` is different from `git checkout <commit ID>` because `git checkout` returns the files not yet committed within the local repository to a previous state, whereas `git revert` reverses changes committed to the local and project repositories.

These are the steps and explanations to use `git revert` for Jennifer. For the first step we only have the explanation. What is the command that should be used in this step?

1. _____ # Look at the git history of the project to find the commit ID
2. Copy the ID (the first few characters of the ID, e.g. 0b1d055).
3. git revert [commit ID]
4. Type in the new commit message.
5. Save and close

▾ Answer:

The command `git log` lists project history with commit IDs.

The command `git show HEAD` shows changes made at the latest commit, and lists the commit ID; however, Jennifer should double-check it is the correct commit, and no one else has committed changes to the repository.

Exercise

What is the output of the last command in

```
cd projx
echo "Title" > abstract.txt
git add abstract,txt
echo "Here we report on the first results of proj X." >> abstract.txt
git commit -m "Adds abstract"
git checkout HEAD abstract.txt
cat abstract.txt #this will print the contents of abstract.txt to the screen
```

1. Here we report on the first results of proj X.
2. Title
3. ◦ Title
   ◦ Here we report on the first results of proj X.
4. Error because you have changed abstract.txt without committing the changes

▾ Answer:

The answer is 2.

The command `git add abstract.txt` places the current version of `abstract.txt` into the staging area. The changes to the file from the second echo command are only applied to the working copy, not the version in the staging area.

So, when `git commit -m "Adds abstract"` is executed, the version of `abstract.txt` committed to the repository is the one from the staging area and has only one line.

At this time, the working copy still has the second line (and git status will show that the file is modified). However, `git checkout HEAD abstract.txt` replaces the working copy with the most recently committed version of credentials.txt.

So, `cat abstract.txt` will output

- Title

Getting Rid of staged changes

`git checkout` can be used to restore a previous commit when unstaged changes have been made, but will it also work for changes that have been staged but not committed? Make a change to `README.md`, add that change, and use `git checkout -- README.md` to see if you can remove your change.

You will notice that `git checkout -- README.md` won't undo the last line we just added.

To unstage `README.md` and keep your last change (the added line):

```
git reset README.md
```

Then you can use `git checkout` to discard the last change and turn `README.md` to its version in the last commit:

```
git checkout -- README.md
```

Explore and Summarize Histories

Exploring history is an important part of Git, and often it is a challenge to find the right commit ID, especially if the commit is from several months ago.

Imagine the project X has more than 50 files. You would like to find a commit that modifies some specific text in `README.md`. When you run `git log`, a very long list appears. How can you narrow down the search?

Recall that the `git diff` command allows us to explore one specific file, e.g., `git diff README.md`. We can apply a similar idea here.

```
git log README,md
```

Unfortunately some of these commit messages are very ambiguous, e.g., `update files`. How can you search through these files?

Both `git diff` and `git log` are very useful and they summarize a different part of the history for you. Is it possible to combine both? Let's try the following:

```
git log --patch README.md
```

You should get a long list of output, and you should be able to see both commit messages and the difference between each commit.

Exercise

What does the following command do?

```
git log --patch HEAD~9 *.txt
```

Key points:

- `git diff` displays differences between commits
- `git checkout` recovers old versions of files

Ignoring unrelated files and folders

The main purpose of Git is to track changes of simple text files, like READMEs or programming scripts. It is not

meant for tracking changes of big and/or binary files, like big NGS-data input files or big pictures. Avoid putting these into a Git repository. Another type of files you should never put into a Git repository is text files with logins and passwords. Once in a Git repository, which you maybe want to make public later, it is almost impossible to remove such sensitive data again.

What if we have files that we do not want Git to track for us, like backup files created by our editor or intermediate files created during data analysis? Let's create a few dummy files:

```
mkdir results
touch results/a.out results/b.out
touch credentials_cbg.txt credentials_tud.txt credentials_other.txt
```

```
git status

# Output
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        credentials_cbg.txt
        credentials_other.txt
        credentials_tud.txt
        results/

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

Putting these files under version control would be a waste of disk space and a security risk. What's worse, having them all listed could distract us from changes that actually matter, so let's tell Git to ignore them.

We do this by creating a file in the root directory of our project called `.gitignore`:

```
vim .gitignore
cat .gitignore

# Output
credentials_*.txt
results/
```

These patterns tell Git to ignore any file whose name ends in `.txt` and starts with `credentials_` and everything in the folder `results`. (If any of these files were already being tracked, Git would continue to track them.)

Once we have created this file, the output of git status is much cleaner:

```
git status

# Output
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

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

The only thing Git notices now is the newly-created `.gitignore` file. You might think we wouldn't want to track it, but everyone we're sharing our repository with will probably want to ignore the same things that we're ignoring. Let's add and commit `.gitignore`:

```
git add .gitignore
git commit -m "Adds .gitignore"
git status

# Output
On branch master
nothing to commit, working tree clean
```

As a bonus, using `.gitignore` helps us avoid accidentally adding files to the repository that we don't want to track:

```
git add credentials_other.txt

# Output
The following paths are ignored by one of your .gitignore files:
projx/credentials_other.txt
Use -f if you really want to add them.
```

If we really want to override our ignore settings, we can use git add `-f` to force Git to add something. For example, `git add -f credentials_other.txt`. We can also always see the status of ignored files if we want:

```
git status --ignored

# Output
On branch master
Ignored files:
  (use "git add -f <file>..." to include in what will be committed)
        credentials_cbg.txt
        credentials_other.txt
        credentials_tud.txt
        results/

nothing to commit, working tree clean
```

Exercise

Given a directory structure that looks like:

```
results/plots
results/data
```

How would you ignore only `results/plots` and not `results/data`?

▾ Answer:

If you only want to ignore the contents of `results/plots`, you can change your `.gitignore` to ignore only the `/plots/` subfolder by adding the following line to your `.gitignore`:

```
results/plots/
```

This line will ensure only the contents of `results/plots` is ignored, and not the contents of `results/data`.

Exercise

How would you ignore all `.txt` files in your root directory except for `final.txt`? Hint: Find out what `!` (the exclamation point operator) does.

▾ Answer:

You would add the following two lines to your `.gitignore`:

```
*.txt          # ignore all txt files
!final.txt     # except final.txt
```

The exclamation point operator will include a previously excluded entry.

Note also that when you've previously committed `.txt` files to the respository they will not be ignored with this new rule. Only future additions of `.txt` files added to the root directory will be ignored.

Exercise

Assuming you have an empty `.gitignore` file, and given a folder structure that looks like:

```
results/data/position/gps/a.dat
results/data/position/gps/b.dat
results/data/position/gps/c.dat
results/data/position/gps/info.txt
results/plots
```

What's the shortest `.gitignore` rule you could write to ignore all .dat files in `result/data/position/gps` ? Do not ignore the `info.txt` .

▾ Answer:

Appending `results/data/position/gps/*.dat` will match every file in `results/data/position/gps` that ends with `.dat` . The file `results/data/position/gps/info.txt` will not be ignored.

Exercise

Given a `.gitignore` file with the following contents:

```
*.dat
!*.dat
```

What will be the result?

▾ Answer:

The `!` modifier will negate an entry from a previously defined ignore pattern. Because the `!*.dat` entry negates all of the previous `.dat` files in the `.gitignore` , none of them will be ignored, and all `.dat` files will be tracked.

Exercise

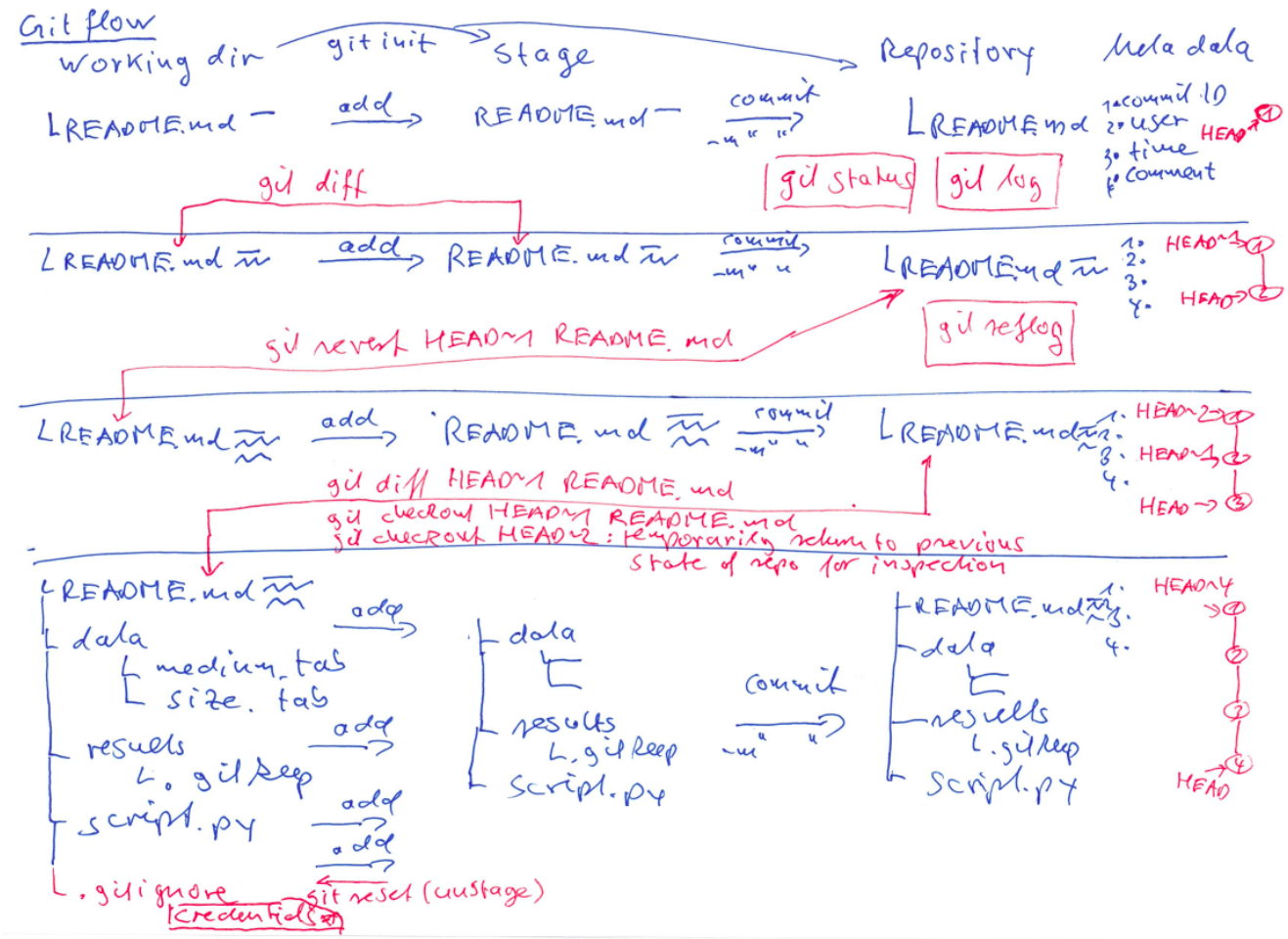
You wrote a script that creates many intermediate log-files of the form `log_01` , `log_02` , `log_03` , etc. You want to keep them but you do not want to track them with Git.

1. Write one `.gitignore` entry that excludes files of the form `log_01` , `log_02` , etc.
2. Test your ignore-pattern by creating some dummy files of the form `log_01` , etc.
3. You find that the file `log_01` is very important after all, add it to the tracked files without changing the `.gitignore` again.
4. Discuss with your neighbor what other types of files could reside in your directory that you do not want to track and hence you would exclude via `.gitignore` .

Key points:

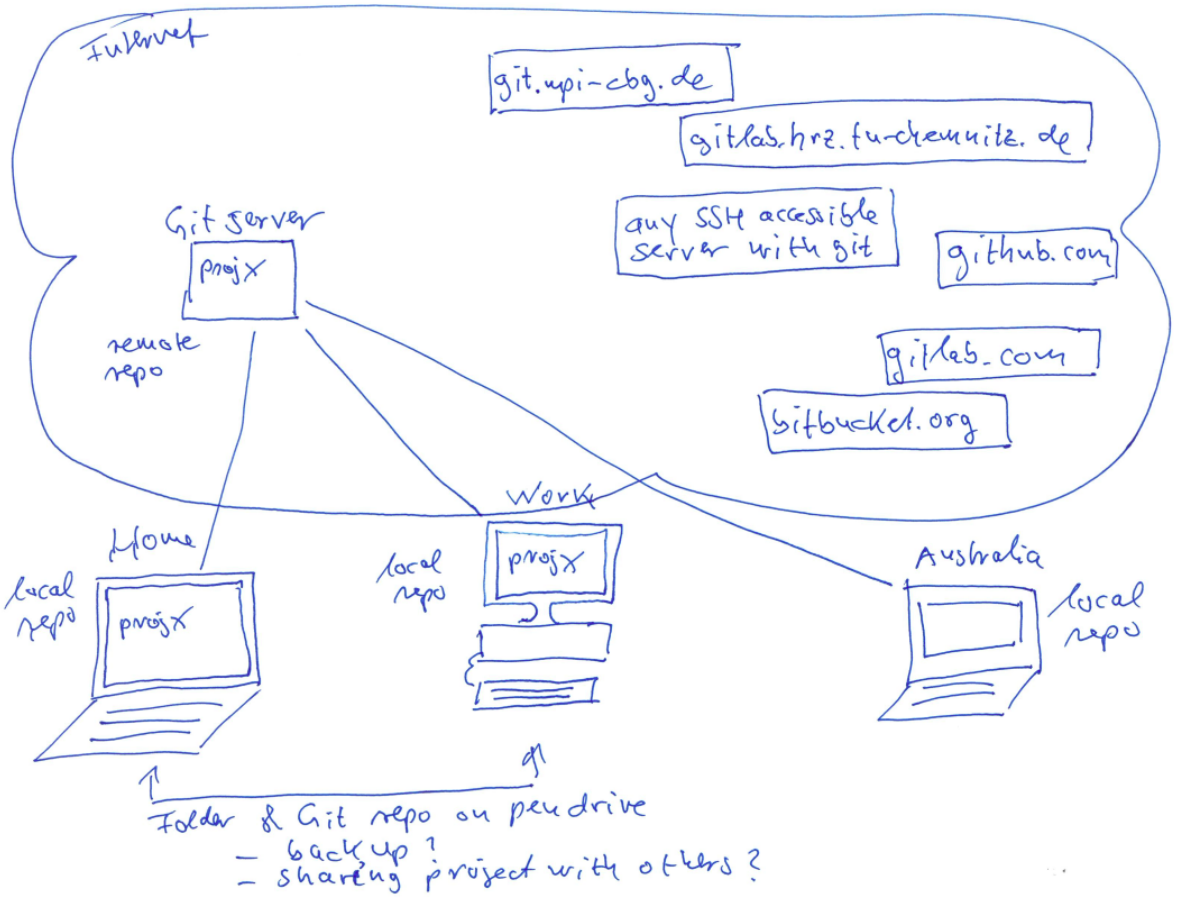
- The `.gitignore` file tells Git what files to ignore

Part 1 in a nutshell: Basics of Git



Part 2: Collaboration and GitLab

Distributed Git repositories



When you have access to a Git server on the internet, then you can have several copies of your Git repository distributed on different computers, eg. on your work computer and home computer. This allows you to share your files, work progress, and project history across these different computers and across different people, eg. collaborators or the world.

The one Git repository on the Git server is usually called remote repository. In most cases there will be exactly one remote repository per project. The Git repositories on your computers are usually called local repositories. Local repositories need to be connected (configured) to know about the remote repository they should exchange data with. There are severl ways to do so and we will exercise two different ways in the sequel.

The CBG/TUD Git server

Let's try to access our Git servers for creating a remote repository.

- git.mpi-cbg.de
- gitlab.hrz.tu-chemnitz.de

Create a remote repository

1. Login
2. Browse around; explore a bit what is there, eg. profile settings
3. Create a new remote **blank project** called `projx`. **Important**: Uncheck box `Initialize repository with a README` before clicking `create project`.

When you create a repository on Gitlab then this is what happens in the background

```
mkdir projx
cd projx
git init
```

Important: Here we created a blank repository without `README.md` on the GitLab server because we want to connect this repository later to our local repository in folder `projx`. If you don't want to connect the remote repository to any pre-existing local repository you can and probably should keep the box checked and let GitLab create a inital version of a README file for you.

Connect your local repository to the remote repository

1. Copy the HTTPS url for your remote repository from the Gitlab website
2. On your computer, navigate into the folder `projx` which is where you have the local repository

```
cd projx

# Take the https address from GitLab where you navigate to your project x and click blue button 'Clone HTTPS'
git remote add origin https://git.mpi-cbg.de/<your_login>/projx
```

We have just let know our local Git repository the internet address of the remote Git repository on the GitLab server.

Some more details about remote repositories

In this episode and the previous one, our local repository has had a single `remote`, called `origin`. A remote is a copy of the repository that is hosted somewhere else, that we can push to and pull from, and there is no reason that you have to work with only one. For example, on some large projects you might have your own copy in your own Gitlab account (you'd probably call this origin) and also the main `upstream` project repository (let's call this upstream for the sake of examples). You would pull from upstream from time to time to get the latest updates that other people have committed.

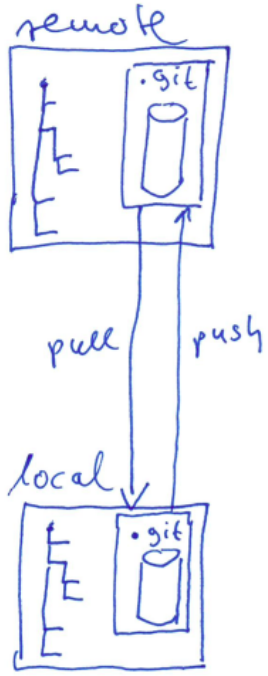
Remember that the name you give to a remote only exists locally. It's an alias that you choose - whether origin, or upstream, or fred - and not something intrinstic to the remote repository.

The git remote family of commands is used to set up and alter the remotes associated with a repository. Here are some of the most useful ones:

1. `git remote -v` lists all the remotes that are configured (we already used this in the last episode)
2. `git remote add [name] [url]` is used to add a new remote
3. `git remote remove [name]` removes a remote. Note that it doesn't affect the remote repository at all - it just removes the link to it from the local repo.
4. `git remote set-url [name] [newurl]` changes the URL that is associated with the remote. This is useful if it has moved, e.g. to a different Gitlab account, or from Gitlab to a different hosting service. Or, if we made a typo when adding it.
5. `git remote rename [oldname] [newname]` changes the local alias by which a remote is known - its name. For

example, one could use this to change upstream to fred.

Update the remote repository to the latest state of your local repository



In your local repository we have all the files we added to it so far. The remote repository is not up-to-date with our local repository. We use `git push` to update the remote repository, ie. `push` our local repository to the remote. Bear in mind that only the Git repository, ie. all that is saved in the hidden folder `.git` will be sent to remote. Local files that are not yet under Git version control will not be sent.

```
cd projx

touch credentials.txt

# To see that file credentials.txt is still untracked
git status

# Because this is the first time we push to the remote repository,
# we use these two push commands to push all contents of our local Git repository
# to the remote Git repository
```

```
git push -u origin --all
git push -u origin --tags


# Output
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 213 bytes | 213.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://git.mpi-cbg.de/<your_login>/projx.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.


# Later, it is usually sufficient to run only
git push

# Output
Everything up-to-date
```

Go to the remote repository on Gitlab and check how it looks now.

If the network you are connected to uses a proxy, there is a chance that your last command failed with “Could not resolve hostname” as the error message. To solve this issue, you need to tell Git about the proxy:

```
git config --global http.proxy http://user:password@proxy.url
git config --global https.proxy https://user:password@proxy.url
```

When you connect to another network that doesn’t use a proxy, you will need to tell Git to disable the proxy using:

```
git config --global --unset http.proxy
git config --global --unset https.proxy
```

Gitlab GUI

Browse to your `projx` repository on Gitlab. Under the `Commits` tab that says “XX commits” (where “XX” is some number). Hover over, and click on, the three buttons to the right of each commit. What information can you gather/explore from these buttons? How would you get that same information in the shell?

The left-most button (with the picture of a clipboard) copies the full identifier of the commit to the clipboard. In the shell, git log will show you the full commit identifier for each commit.

When you click on the middle button, you’ll see all of the changes that were made in that particular commit. Green shaded lines indicate additions and red ones removals. In the shell we can do the same thing with git diff. In

particular, git diff ID1..ID2 where ID1 and ID2 are commit identifiers (e.g. git diff a3bf1e5..041e637) will show the differences between those two commits.

The right-most button lets you view all of the files in the repository at the time of that commit. To do this in the shell, we'd need to checkout the repository at that particular time. We can do this with git checkout ID where ID is the identifier of the commit we want to look at. If we do this, we need to remember to put the repository back to the right state afterwards!

Exercise

How does GitLab displays time stamps of commits, and why?

▾ Answer:

Gitlab displays timestamps in a human readable relative format (i.e. "22 hours ago" or "three weeks ago"). However, if you hover over the timestamp, you can see the exact time at which the last change to the file occurred.

Exercise

In this episode, we introduced the "git push" command. How is "git push" different from "git commit"?

▾ Answer:

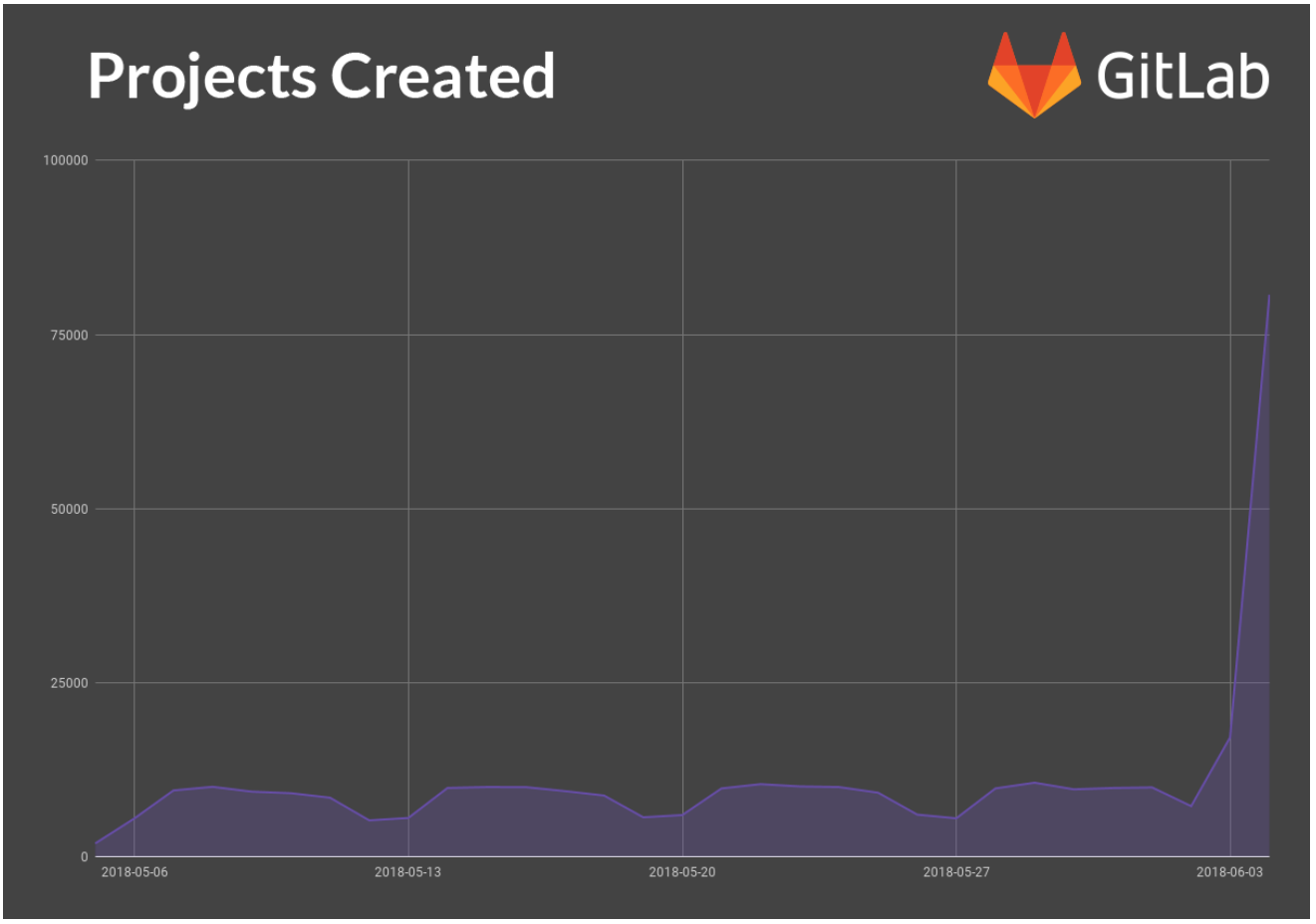
When we push changes, we're interacting with a remote repository to update it with the changes we've made locally (often this corresponds to sharing the changes we've made with others). Commit only updates your local repository.

Direct uploading files into the remote repository

You can add files to the remote repository by:

1. adding them to your local repository and then push the changes in the local repository to the remote
2. directly upload files into the remote repository

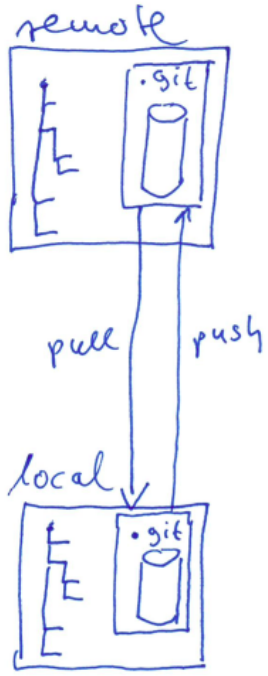
Download and save the plot with title `Projects created` from https://about.gitlab.com/blog/2018/06/03/movingtogitlab/ to your computer. The downloaded file is called `projects-created.png` and it looks like:



Go to your remote repository `projx` and:

1. create a sub folder with name `graphics` and upload the file `projects-created.png` into it
2. add a file `LICENSE` using the button on the main page of your repository Gitlab offers you for this

Update your local repository with the latest changes from the remote respository



On your computer go into the folder `projx`

```
cd projx

git pull

ls  # check which files you have now locally

git log  # check the commit messages of the latest commits
```

README.md: Gitlab flavored Markdown

The file we created `README.md` is a simple text file. It is displayed by Gitlab by default on the main page of the remote respository. You can pimp-up it's appearance by using special `code` to define headline, draw horizontal lines, add graphics.

https://docs.gitlab.com/ee/user/markdown.html

Some examples:

```
# Headline
- key point
- another key point
- another key point

## Sub headline
1. key point 1
2. key point 2
3. key point 3

This adds the graphic we added to the remote repository to sub folder `graphics`
![](graphics/projects-created.png)
```

Key points:

- A local Git repository can be connected to one or more remote repositories.
- git push copies changes from a local repository to a remote repository.
- git pull copies changes from a remote repository to a local repository.

Collaborating

Let us simulate collaboration with one collaborator. Although this section is on collaborating, you will be the two `parties` involved in this collaboration.

Grant access to your remote respository

You are the owner of remote repositories you have create. You have all permissions, read (pull) and write (push). If you project is private then no other user can access it. You need to invite to you repository collaborators as members or collaborators. Let's check out how this would work on GitLab.

On GitLab you can grant access to your private repositories by navigating to the outer-left pannel: `Project information and then Members`.

As you have both roles: you and your collaborator, you don't need to invite yourself to you remote repository here.

Collaborators need to get a local copy of the remote repository

To collaborate with you via your remote repository collaborators need to get a local copy of the remote repository on their computer. This is done with `git clone URL`

Open a second terminal window. This window will represent your collaborator, working on another computer. Navigate into folder `git_course` and clone the remote repository to your computer

```
cd git_course
git clone https://git.mpi-cbg.de/<your_login>/projx.git projx2
```

Two important things:

1. You get the URL for cloning from the GitLab main page of your project
2. Here you must tell git to put the local copy into a folder with name `projx2`. If you don't specify this folder, than Git will try to put the copy automatically into folder projx where we have already a local repository

Git add remote vs git clone

Remember that we added with `git add remote` the information about the remote repository to the local repository in `projx`. Another and easier way to get a local repository readily configured to connect to a specific remote repository is to:

1. Create first the remote respository on Gitlab
2. Clone it to your local computer with `git clone`

This way is often more handy and quicker. So, in most cases when you start a new project which you want to have under Git version control, first create the remote repository on GitLab and then clone it to you computer.

Key points:

- `git clone` copies a remote repository to create a local repository with a remote called origin automatically set up.

Get files and changes from your collaborator

You (the collaborator) now adds a file to the repository in folder `projx2`

```
cd projx2

vim ideas.txt
cat ideas.txt

# Output
We should do x first and then y.

git add ideas.txt
git commit -m 'adds project ideas'
git push
```

Now the two repositories (collaborator's local, and remote repository on Gitlab) are in sync, but your (you) local repository is behind, ie. it misses the latest changes.

Now, you (you) want to get the contributions from your collaborator to you local repository in folder `projx`

```
cd projx
git pull
```

Now the three repositories (your local, collaborator's local, and remote repository on Gitlab) are all in sync.

Exercise

The owner pushed commits to the repository without giving any information to the collaborator. How can the collaborator find out what has changed with command line? And on Gitlab?

▾ Answer:

On the command line, the collaborator can use `git fetch origin main` to get the remote changes into the local repository, but without merging them. Then by running `git diff main origin/main` the collaborator will see the changes output in the terminal.

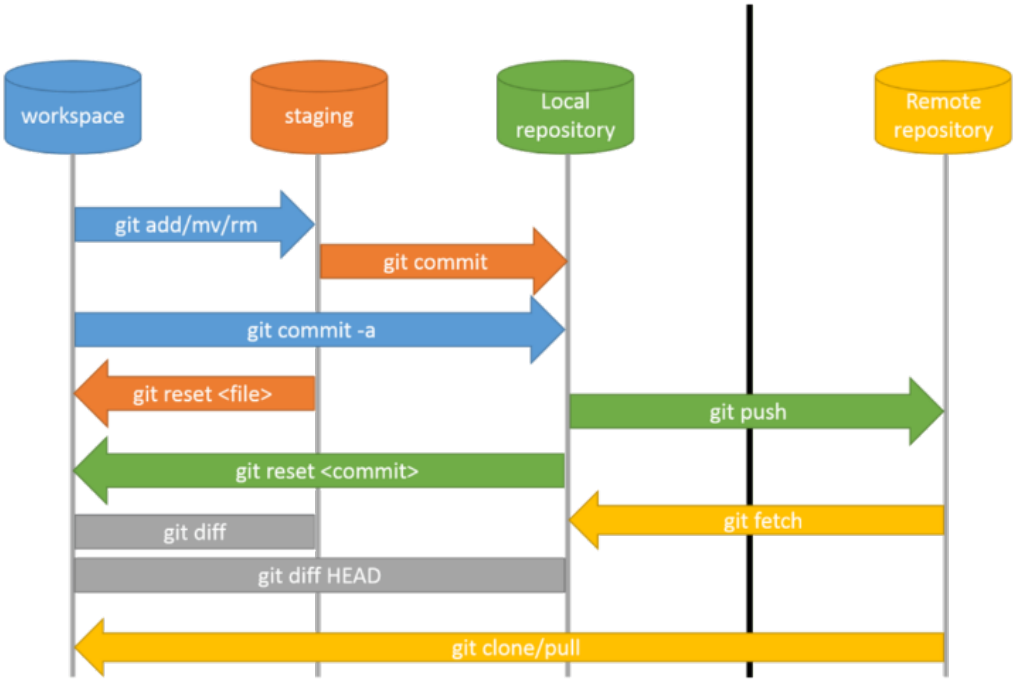
On Gitlab, the collaborator can go to the repository and click on `commits` to view the most recent commits.

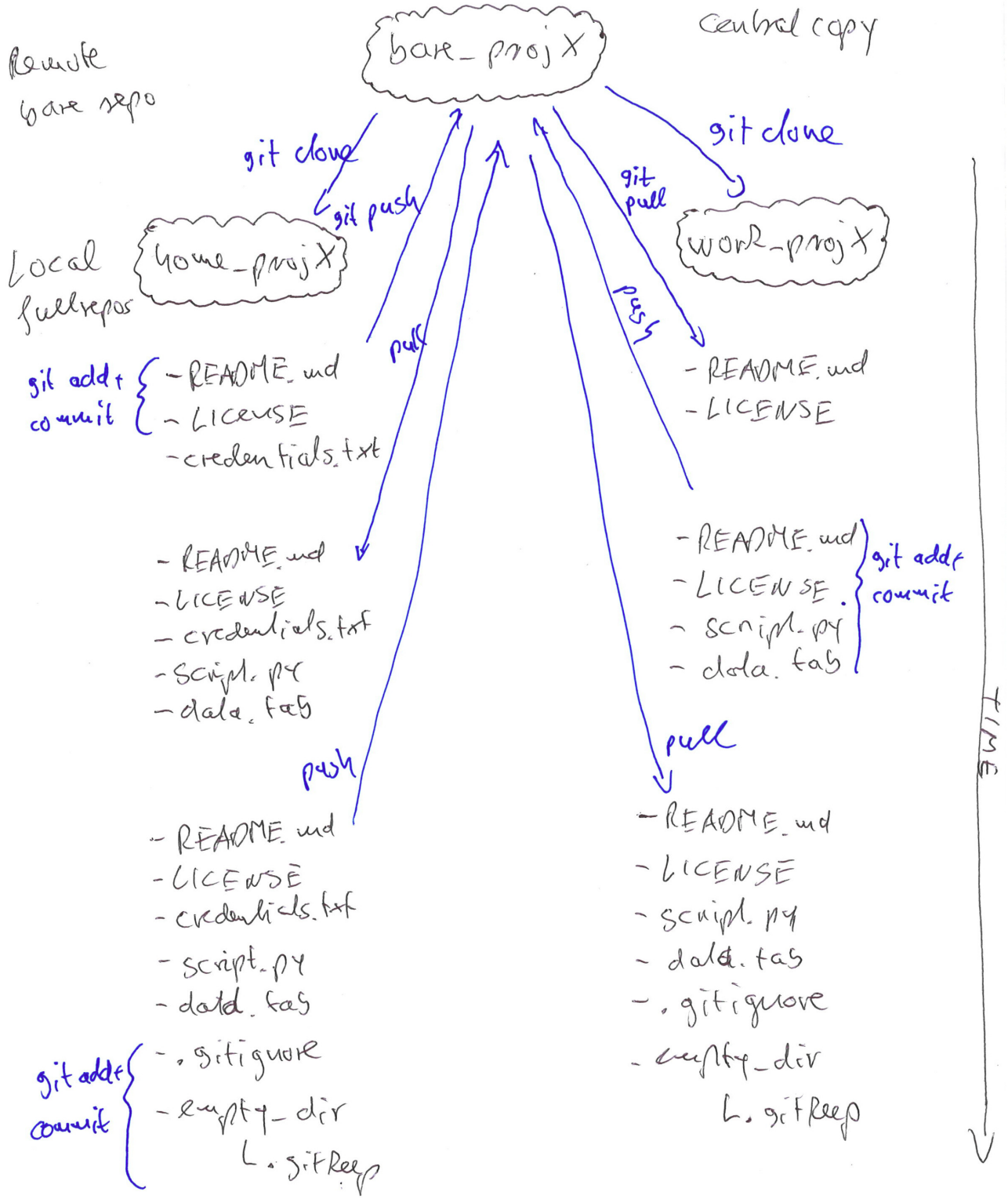
Basic collaborative workflow

In practice, it is good to be sure that you have an updated version of the repository you are collaborating on, so you should git pull before making our changes. The basic collaborative workflow would be:

1. update your local repo with git pull origin main
2. make your changes and stage them with git add,
3. commit your changes with git commit -m, and
4. upload the changes to GitHub with git push origin main

It is better to make many commits with smaller changes rather than of one commit with massive changes: small commits are easier to read and review.


workspace
staging
Local repository
Remote repository
git add/mv/rm
git commit
git commit -a
git reset <file>
git push
git reset <commit>
git fetch
git diff
git diff HEAD
git clone/pull


Remote bare repo
bare_projX
Central copy
git clone
git push
git clone
git pull
push
Local fullrepos
home_projX
work_projX
git add + commit
- README.md
- LICENSE
- credentials.txt
- README.md
- LICENSE
pull
- README.md
- LICENSE
- credentials.txt
- script.py
- data.tab
- README.md
- LICENSE
- script.py
- data.tab
git add + commit
push
pull
- README.md
- LICENSE
- credentials.txt
- script.py
- data.tab
- .gitignore
- empty_dir
└ .gitKeep
git add + commit
- README.md
- LICENSE
- script.py
- data.tab
- .gitignore
- empty_dir
└ .gitKeep
TIME

Conflicts

As soon as people can work in parallel, they'll likely step on each other's toes. This will even happen with a single person: if we are working on a piece of software on both our laptop and a server in the lab, we could make different changes to each copy.

Although nobody likes conflics in real life, conflicts in collaborative programming or writing projects are nothing bad. They happen all the time and the good thing with Git is that Git has many build-in fueatures to help is to resolved conflicts.

To see how we can resolve conflicts, we must first create one. The file `README.md` currently looks like this in both partners' copies of our repository `projx` and `proxj2`:

```
cat README.me

# Output
# Project X
- owner: I
- start date: today
```

Let's add a line in the local copy of the collaborator in folder `projx2`:

```
cd projx2

vim README.md
cat README.me

# Output
# Project X
- owner: I
- start date: today
- line added by collaborator
```

We add this change to the local repository of the collaborator and push the change to Gitlab:

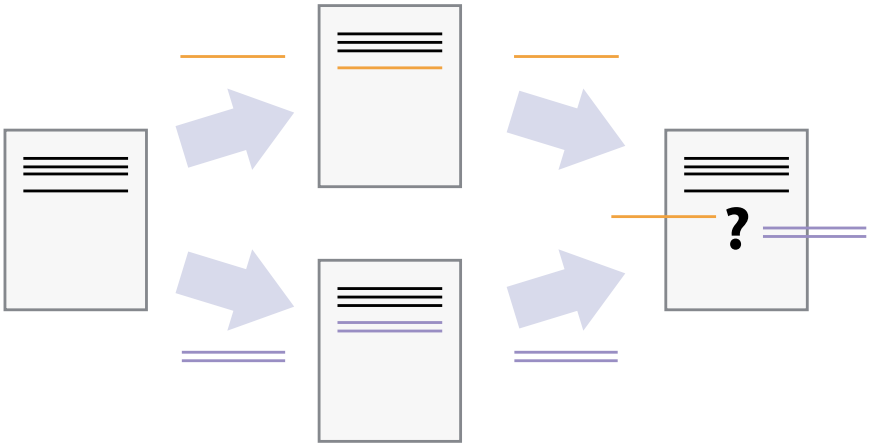
```
git add README.md
git commit -m 'adds line from collaborator'
git push
```

Now let's change `README.md` in our local repository in folder `projx` and add a different change without updating (pull) before from Gitlab:

```
cd projx

vim README.md
cat README.me

# Output
# Project X
- owner: I
- start date: today
- line added by us
```

We can commit the change locally:

```
git add README.md
git commit -m 'adds line from us'
```



Git won't let us push it to Gitlab:

```
git push

# You should see a similar output
To https://gitlab.com/<your_login>/projx.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://gitlab.com/<your_login>/projx.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Git rejects the push because it detects that the remote repository has new updates that have not been incorporated into the local repository. What we have to do is pull the changes from Gitlab, merge them into the copy we're currently working in, and then push that. Let's start by pulling:

```
git pull

# Output similart toremote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://gitlab.com/<your_login>/projx
 * branch            main     -> FETCH_HEAD
    29aba7c..dabb4c8  main     -> origin/main
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

The `git pull` command updates the local repository to include those changes already included in the remote repository. After the changes from remote branch have been fetched, Git detects that changes made to the local copy overlap with those made to the remote repository, and therefore refuses to merge the two versions to stop us from trampling on our previous work. The conflict is marked in in the affected file:

```
cat README.md

# Output
# Project X
- owner: I
- start date: today
<<<<<<< HEAD
- line added by us
```

```
=======
- line added by collaborator
>>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d
```

Our change is preceded by <<<<<<< HEAD. Git has then inserted ======= as a separator between the conflicting changes and marked the end of the content downloaded from Gitkab with >>>>>>>. (The string of letters and digits after that marker identifies the commit we've just downloaded.)

It is now up to us to edit this file to remove these markers and reconcile the changes. We can do anything we want: keep the change made in the local repository, keep the change made in the remote repository, write something new to replace both, or get rid of the change entirely. Let's replace both so that the file looks like this:

```
 cat README.md

 # Output
 # Project X
- owner: I
- start date: today
- new line resulting from merging conflicting line
```

To finish merging, we add `README.md` to the changes being made by the merge and then commit:

```
git add README.md
git status

# Output
On branch main
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

        modified:   README.md
```

```
git commit -m "Merge changes from Gitlab"
```

Now we can push our changes to Gitlab:

```
git push

# Output similar to
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 4), reused 6 (delta 4), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://gitlab.com/<your_login>/projx
 * branch            main     -> FETCH_HEAD
   dabb4c8..2abf2b1  main     -> origin/main
Updating dabb4c8..2abf2b1
Fast-forward
 README.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Git keeps track of what we've merged with what, so the collaborator won't have to fix things by hand again when pulling latest changes from the remote repository again in folder `projx2`:

```
cd projx2
git pull

# Output similar
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 4), reused 6 (delta 4), pack-reused 0
Unpacking objects: 100% (6/6), done.
```

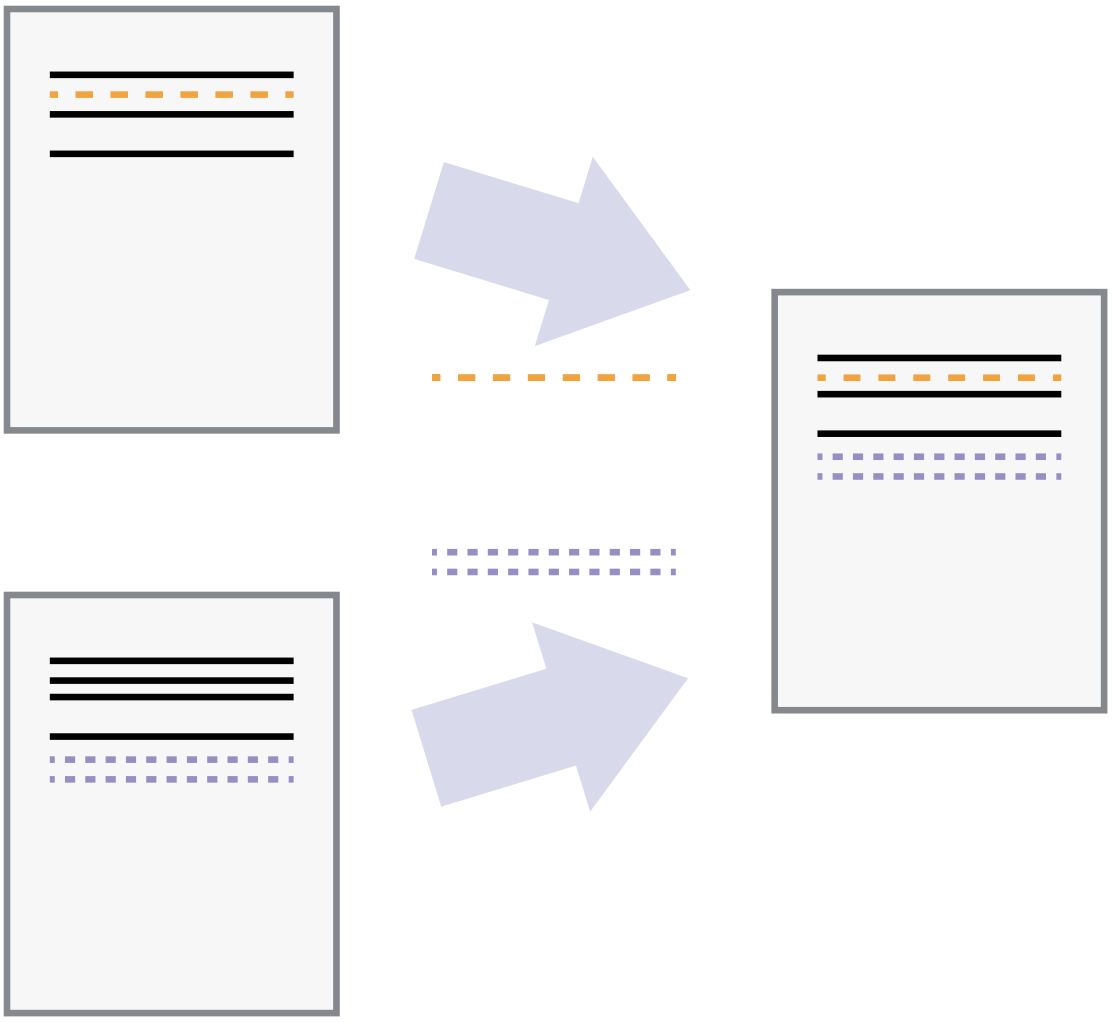
```
From https://gitlab.com/<your_login>/projx
 * branch            main     -> FETCH_HEAD
    dabb4c8..2abf2b1  main     -> origin/main
Updating dabb4c8..2abf2b1
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

The collaborator gets the merged file `README.md` :

```
cat README.md

# Output
 # Project X
- owner: I
- start date: today
- new line resulting from merging conflicting line
```

The collaborator wont’t need to merge again because Git knows someone has already done that.

Git can resolve some conflicts automatically



Reduce risk of conflics

Git's ability to resolve conflicts is very powerful, but conflict resolution costs time and effort, and can introduce errors if conflicts are not resolved correctly. If you find yourself resolving a lot of conflicts in a project, consider these technical approaches to reducing them:

1. Pull from upstream more frequently, especially before starting new work
2. Use topic branches to segregate work, merging to main when complete
3. Make smaller more atomic commits
4. Where logically appropriate, break large files into smaller ones so that it is less likely that two authors will alter the same file simultaneously

Conflicts can also be minimized with project management strategies:

1. Clarify who is responsible for what areas with your collaborators
2. Discuss what order tasks should be carried out in with your collaborators so that tasks expected to change the same lines won't be worked on simultaneously
3. If the conflicts are stylistic churn (e.g. tabs vs. spaces), establish a project convention that is governing and use code style tools (e.g. htmltidy, perltidy, rubocop, etc.) to enforce, if necessary

Conflics in non-textual files

What does Git do when there is a conflict in an image or some other non-textual file that is stored in version control?

Let's try it. In our project `projx` we have already a sub folder graphics.

The collaborator adds another picture there, essentially a dummy random picture.

```
cd projx2
cd graphics
head -c 1024 /dev/urandom > pic.jpg
ls -lh pic.jpg
```

`ls` shows us that this created a 1-kilobyte file. It is full of random bytes read from the special file, `/dev/urandom`.

```
add pic.jpg
git commit -m 'adds a picture pic1'
git push
```

Assume in the meantime we have added a picture with the same file name into the same sub folder:

```
cd projx
cd graphics
head -c 1024 /dev/urandom > pic.jpg
ls -lh pic.jpg
```

```
add pic.jpg
git commit -m 'adds a picture pic1'
git push

# Output similar
To https://gitlab.com/<your_login>/projx.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://gitlab.com/<your_login>/projx.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

We've learned that we must pull first and resolve any conflicts:

```
git pull
```

When there is a conflict on an image or other binary file, git prints a message like this:

```
git pull origin main
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://gitlab.com/<your_login>/projx.git
 * branch            main     -> FETCH_HEAD
   6a67967..439dc8c  main     -> origin/main
warning: Cannot merge binary files: graphics/pic1.jpg (HEAD vs. 439dc8c08869c342438f6dc4a2b615b05b93c76e)
Auto-merging graphics/pic1.jpg
CONFLICT (add/add): Merge conflict in graphics/pic1.jpg
Automatic merge failed; fix conflicts and then commit the result.
```

The conflict message here is mostly the same as it was for `README.md`, but there is one key additional line:

```
warning: Cannot merge binary files: graphics/pic1.jpg (HEAD vs. 439dc8c08869c342438f6dc4a2b615b05b93c76e)
```

Git cannot automatically insert conflict markers into an image as it does for text files. So, instead of editing the image file, we must check out the version we want to keep. Then we can add and commit this version.

On the key line above, Git has conveniently given us commit identifiers for the two versions of `pic1.jpg`. Our version is HEAD, and the collaborator's version is 439dc8c0.... If we want to use our version, we can use git checkout:

```
git checkout HEAD graphics/pic1.jpg
git add graphics/pic1.jpg
git commit -m "Use our picture pic1"
```

If instead we want to use the collaborator's version, we can use `git checkout` with the collaborator's commit identifier, 439dc8c0:

```
git checkout 439dc8c0 graphics/pic1.jpg
git add graphics/pic1.jpg
git commit -m "Use collaborator's picture pic1"
```

We can also keep both images. The catch is that we cannot keep them under the same name. But, we can check out each version in succession and rename it, then add the renamed versions. First, check out each image and rename it:

```
git checkout HEAD graphics/pic1.jpg
git mv graphics/pic1.jpg graphics/pic1_me.jpg
git checkout 439dc8c0 graphics/pic1.jpg
mv graphics/pic1.jpg graphics/pic1_collaborator.jpg
```

Then, remove the old `graphics/pic1.jpg` from the Git repository and add the two new files:

```
git rm graphics/pic1.jpg
git add graphics/pic1_me.jpg
git add graphics/pic1_collaborator.jpg
git commit -m "Use two image versions: pic_me and pic_collaborator"
```

Now both images are checked into the repository, and `graphics/pic1.jpg` no longer exists. We have resolved the conflict by keeping both versions of this picture.

```
git push
```

Exercise

You sit down at your computer to work on a shared project that is tracked in a remote Git repository. During your work session, you take the following actions, but not in this order:

- Make changes by appending the number 100 to a text file numbers.txt
- Update remote repository to match the local repository
- Celebrate your success with some fancy beverage(s)
- Update local repository to match the remote repository
- Stage changes to be committed
- Commit changes to the local repository

In what order should you perform these actions to minimize the chances of conflicts? Put the commands above in order in the action column of the table below. When you have the order right, see if you can write the corresponding commands in the command column. A few steps are populated to get you started.

Order	Action	Git command
1	_____	_____
2		`echo 100 >> numbers.txt`
3		
4		
5		
6	Celebrate!	AFK

▾ Answer:

Order	Action	Git command
1	Update local	`git pull origin main`
2	Make changes	`echo 100 >> numbers.txt`
3	Stage changes	`git add numbers.txt`
4	Commit changes	`git commit -m "Add 100 to numbers.txt"`
5	Update remote	`git push origin main`
6	Celebrate!	AFK

Further reading and watching

Important: How to work with Git has changed over time. Rely on latest tutorials. Eg. `git reset` should probably not be used anymore?

- Short intro
  - https://www.youtube.com/watch?v=uUuTYDg9XoI
- Git workflow
  - https://www.youtube.com/watch?v=3a2x1iJFJWc
- Long intro
  - https://www.youtube.com/watch?v=8JJ101D3knE
  - missing: browsing history, branching&merging, collaboration, rewriting history
- Introduction Git+Gitlab
  - https://www.youtube.com/watch?v=4lxvVj7wlZw
- SSH-Key
  - https://www.youtube.com/watch?v=iXuIp5uNnLk&list=RDCMUCUUl_HXJjU--iYjUklgEcTw&index=2
- Repos+Fetch+Pull:
  - https://www.youtube.com/watch?v=3a2x1iJFJWc
- Example for how to coordinate work with branches for software development:
  - https://www.youtube.com/watch?v=ykZbBD-CmP8
- Undo changes+reset+clean+revert (but may be outdated; should reset be used anymore?):
  - https://www.youtube.com/watch?v=FdZecVxzJbk
  - https://geekflare.com/git-reset-vs-revert-vs-rebase/
- Git without console: GitGraph in VS Code
  - https://www.youtube.com/watch?v=foXiEpYA08A
- Git remotes, remote tracking branches, forks, remote branches
  - https://www.youtube.com/watch?v=Gg4bLk8cGNo
- Git bare repos (local)
  - https://www.youtube.com/watch?v=8aZW9mYOxhc
- Git book
  - https://git-scm.com/book/en
- Cool way to manage your dotfiles with a bare git repo:
  - https://www.youtube.com/watch?v=tBoLDpTWVOM
- Git course from Software Carpentries
  - https://swcarpentry.github.io/git-novice