

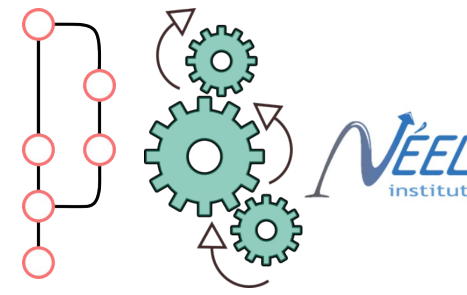
# Git Workshop

Neel workshop

13/11/25

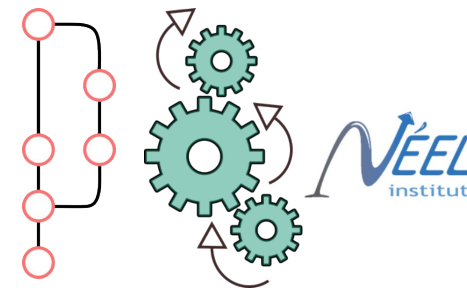
Those slides can also be seen at  
<https://wiki.estaca.net>, then click on "Git Workshop"

# Layout

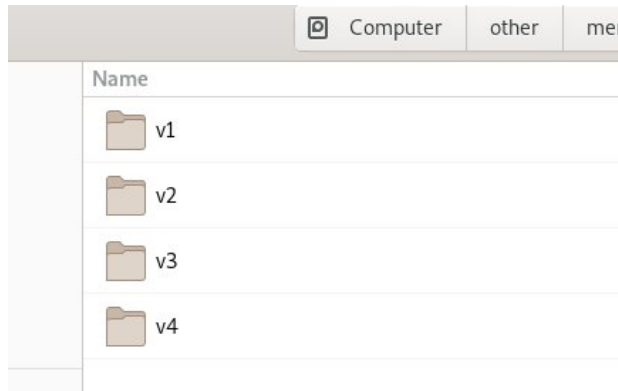


- What is Git and why do I need it?
- Git basics: the three trees
- Branches: why and how?
- Conflict solving
- Exercise 1: local branch merging and conflict solving
- Using a remote repository
- Exercise 2: collaborative editing with a remote repository

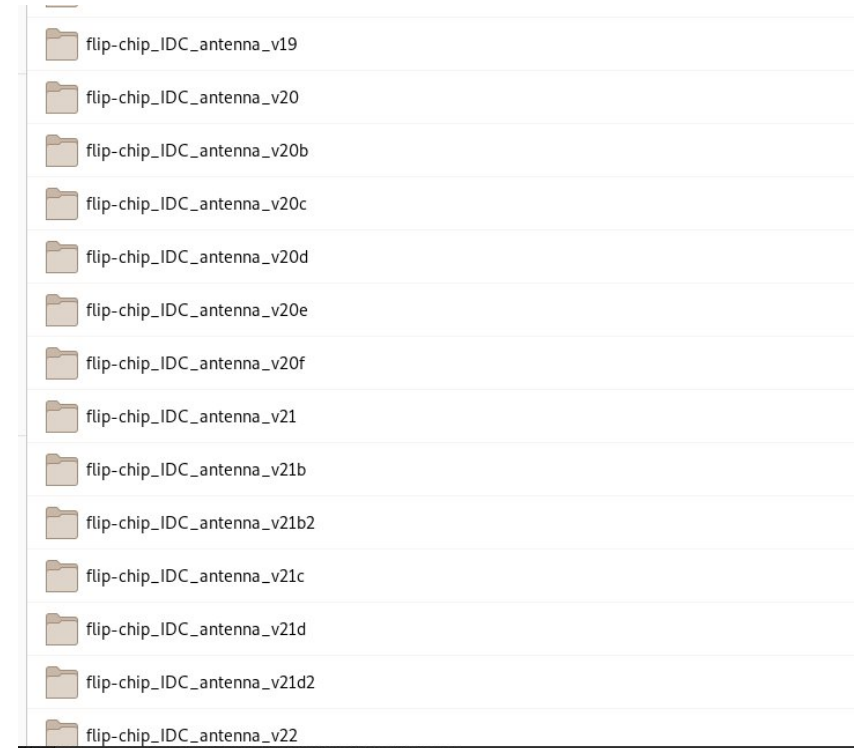
# What is Git and why do I need it ?



- Git is a Version Control System (VCS): it is used to keep tracks of changes you make to a given folder

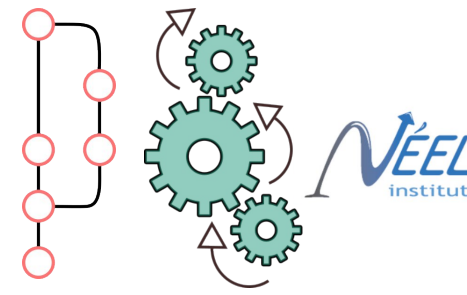


This is bad

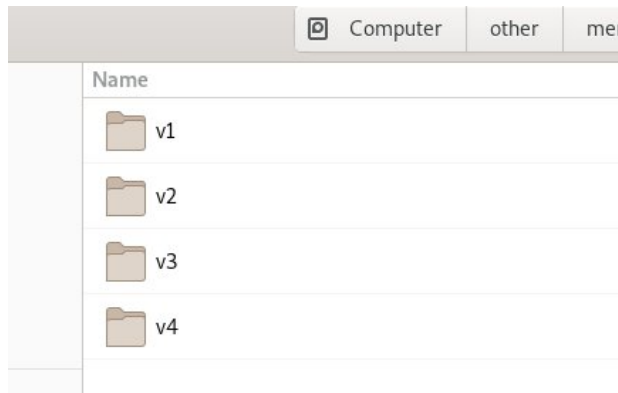


This is even worse

# What is Git and why do I need it ?



- Git is a Version Control System (VCS): it is used to keep tracks of changes you make to a given folder

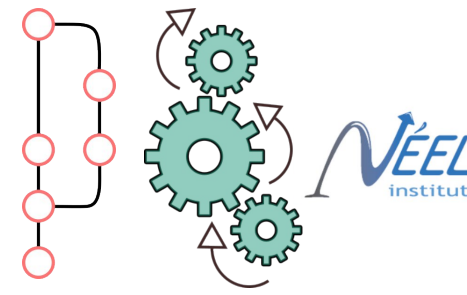


What you want is usually to:

1. have the most recent state of your project only
2. While keeping the possibility to access the history if needed, and/or go back in time, change stuffs, etc.

**Git does that, and much more: it is used to do collaborative work, when many persons are modifying the same folder**

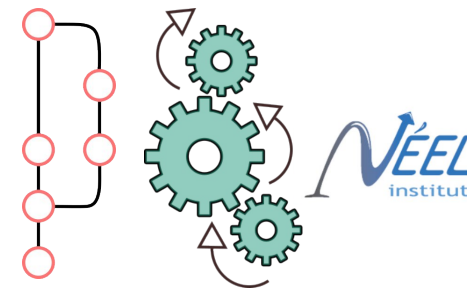
# What is Git and why do I need it ?



- 90% of software development teams use a VCS
- 70% of the teams use Git
- They spend in average 5 hours per day using a VCS
- Git is open source, free and included natively in linux. It is available on all platforms otherwise: <https://git-scm.com/download/>

# Installing Git

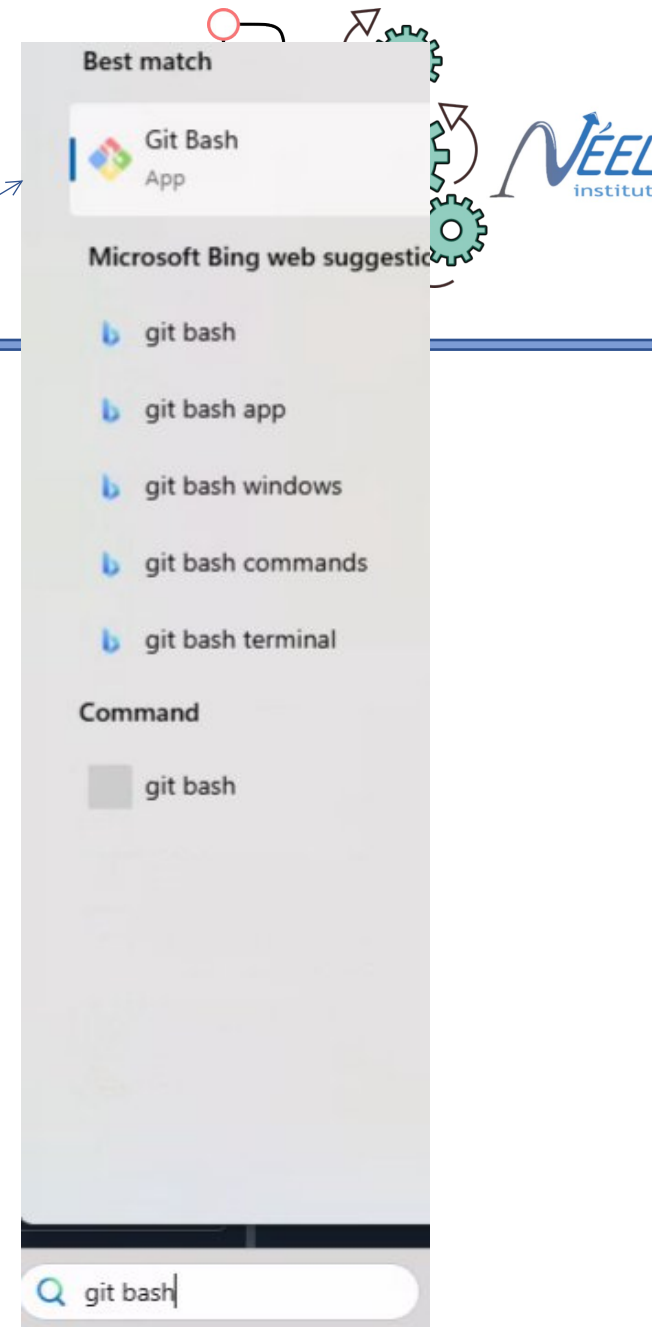
**Free and Open source !!!**



- On linux, you can install it with your package manager (example for Debian/Ubuntu/Mint... : **sudo apt install git** in a terminal)
- On MacOS, install homebrew if not already there (execute in a terminal the command in the website **brew.sh**), then execute **brew install git** in a terminal
- On Windows, install the git-scm software:  
<https://git-scm.com/install/windows>

# Installing Git

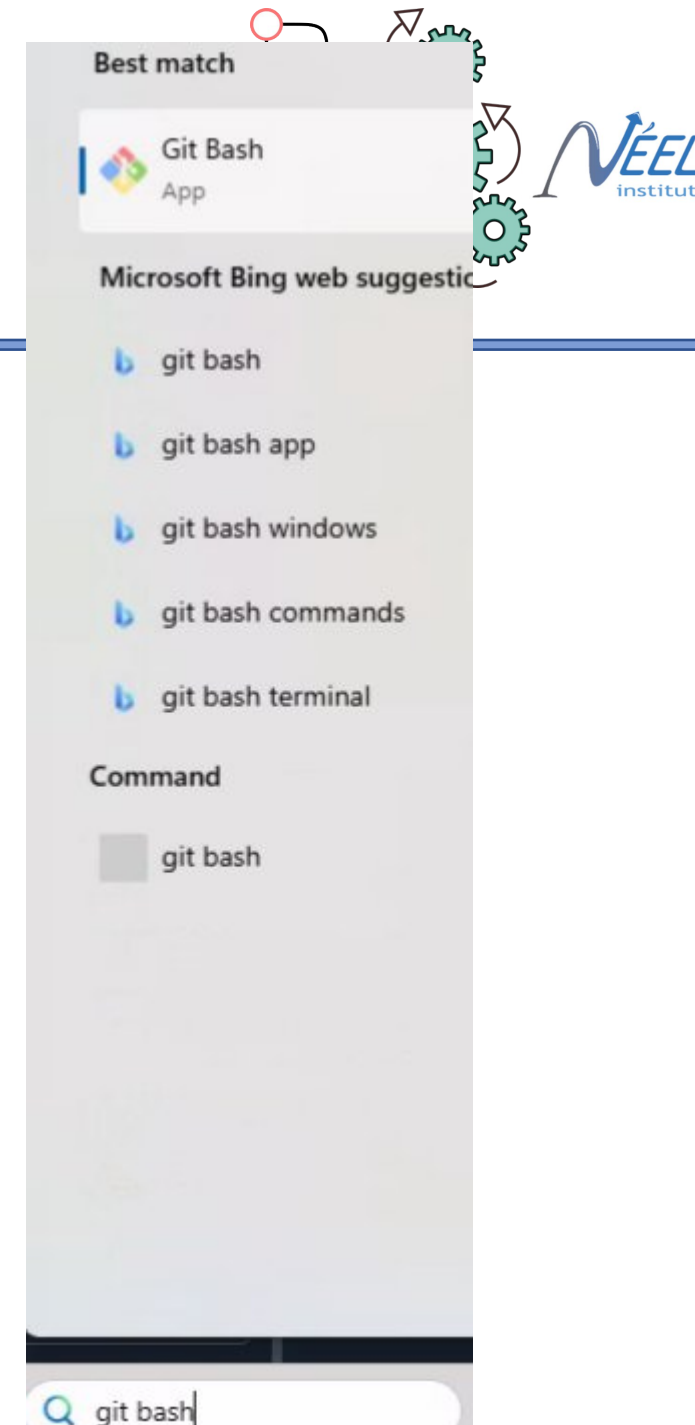
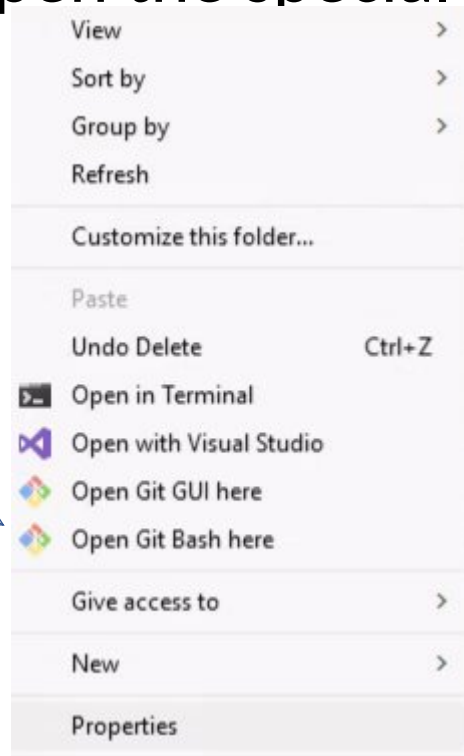
When you are done, open a terminal (for MacOS or Linux), or on windows open the special "Git Bash" terminal



# Installing Git

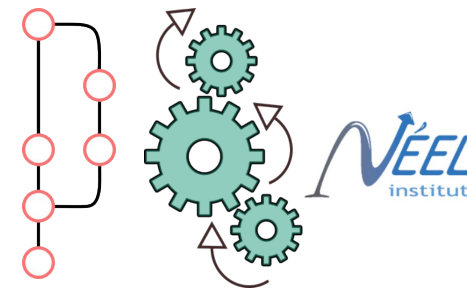
When you are done, open a terminal (for MacOS or Linux), or on windows open the special "Git Bash" terminal

you can also open it from your file explorer after a right click

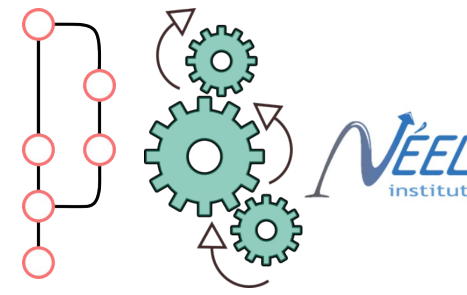




# Git basics: the three trees



# Git basics: the three trees

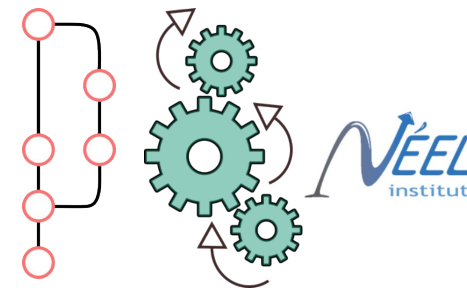


- Everything starts here: This is where you perform changes

working directory



# Git basics: the three trees

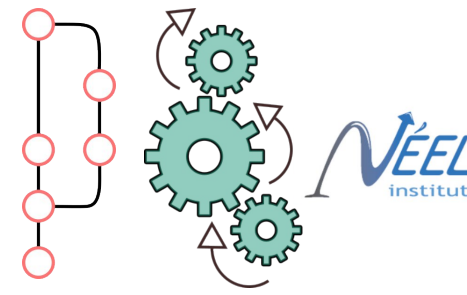


- Everything starts here: This is where you perform changes
- Let's go and start a repository from scratch, from the command line

working directory



# Git basics: the three trees



- Everything starts here: This is where you perform changes
- Let's go and start a repository from scratch, from the command line

working directory



```
~/Documents/Git$ mkdir test1
```

```
~/Documents/Git$ cd test1/
```

```
~/Documents/Git/test1$ nano myfile.txt
```

```
~/Documents/Git/test1$ git init
```

**hint:** Using '**master**' as the name for the initial branch. This default branch name

**hint:** is subject to change. To configure the initial branch name to use in all

**hint:** of your new repositories, which will suppress this warning, **call:**

**hint:**

**hint:** `git config --global init.defaultBranch <name>`

**hint:**

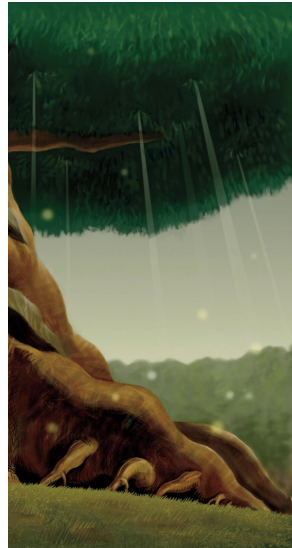
**hint:** Names commonly chosen instead of '**master**' are '**main**', '**trunk**' and

**hint:** '**development**'. The just-created branch can be renamed via this **command:**

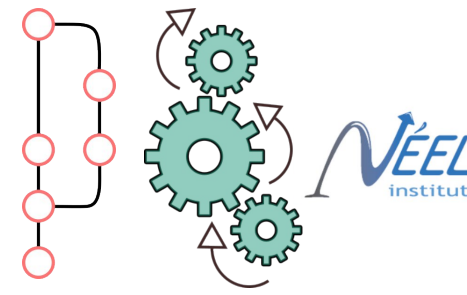
**hint:**

**hint:** `git branch -m <name>`

Initialized empty Git repository in ~/Documents/Git/test1/.git/



# Git basics: the three trees



- Everything starts here: This is where you perform changes
- Let's go and start a repository from scratch, from the command line

working directory



```
~/Documents/Git$ mkdir test1  
~/Documents/Git$ cd test1/  
~/Documents/Git/test1$ nano myfile.txt  
~/Documents/Git/test1$ git init
```

**hint:** Using **'master'** as the name for the initial branch. This default branch name

**hint:** is subject to change. To configure the initial branch name to use in all

**hint:** of your new repositories, which will suppress this warning, **call:**

**hint:**

**hint:** `git config --global init.defaultBranch <name>`

**hint:**

**hint:** Names commonly chosen instead of **'master'** are **'main'**, **'trunk'** and

**hint:** **'development'**. The just-created branch can be renamed via this **command:**

**hint:**

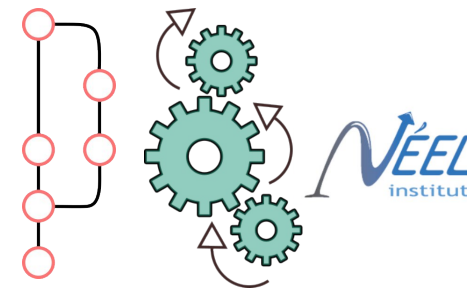
**hint:** `git branch -m <name>`

Initialized empty Git repository in ~/Documents/Git/test1/.git/

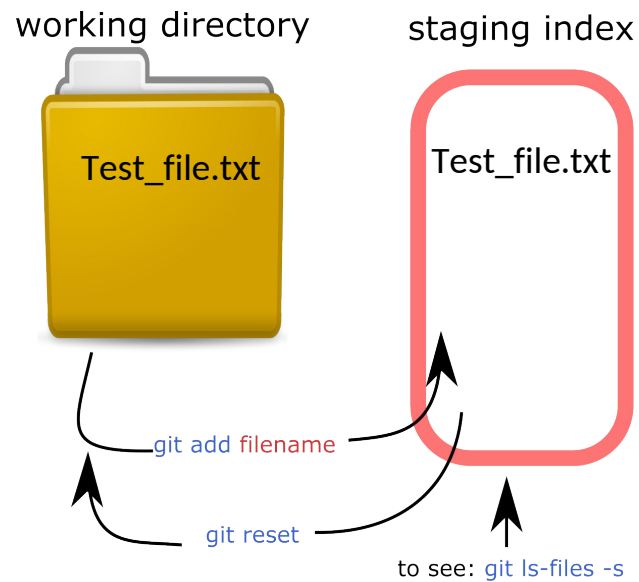
Here I made some changes with a text editor called "nano", but you can use any text editor you want



# Git basics: the three trees



- Now let's put this change in the second tree



to see the difference between the state of the three trees: `git status`  
to see the difference since your last commit: `git diff`

```
~/Documents/Git/test1$ git add myfile.txt
```

```
~/Documents/Git/test1$ git status
```

On branch master

No commits yet

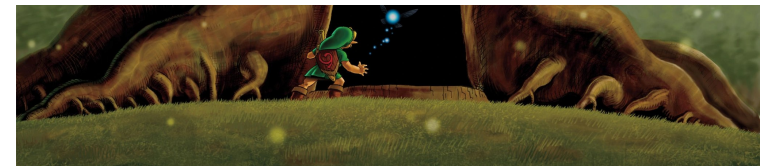
Changes to be **committed**:

(use "`git rm --cached <file>...`" to unstage)    new **file**: myfile.txt

```
~/Documents/Git/test1$ git ls-files -s
```

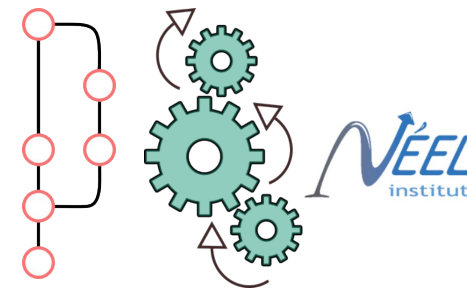
```
100644 ce013625030ba8dba906f756967f9e9ca394464a 0
```

myfile.txt





# Git basics: the three trees



- Now let's put this change in the third tree



git add filename

git reset

```
~/Documents/Git/test1$ git commit -m "initial commit"
```

```
[master (root-commit) 1715f20] initial commit
```

```
1 file changed, 1 insertion(+)  
create mode 100644 myfile.txt
```

```
~/Documents/Git/test1$ git status
```

```
On branch master
```

```
nothing to commit, working tree clean
```

```
~/Documents/Git/test1$ git log --all --graph
```

```
* commit 1715f206ef9ba2cb14ba35e3ae97c9f84e1c56ae (HEAD -> master)
```

```
Author:
```

```
Date: Thu Nov 6 08:31:41 2025 +0100
```

initial commit



to see: `git ls-files -s`

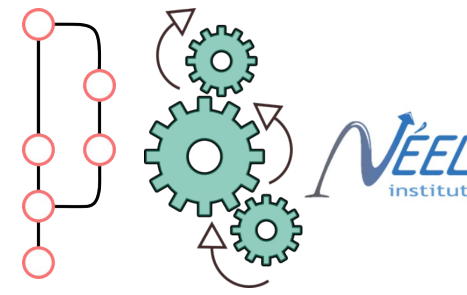


to see: `git log --all --graph`

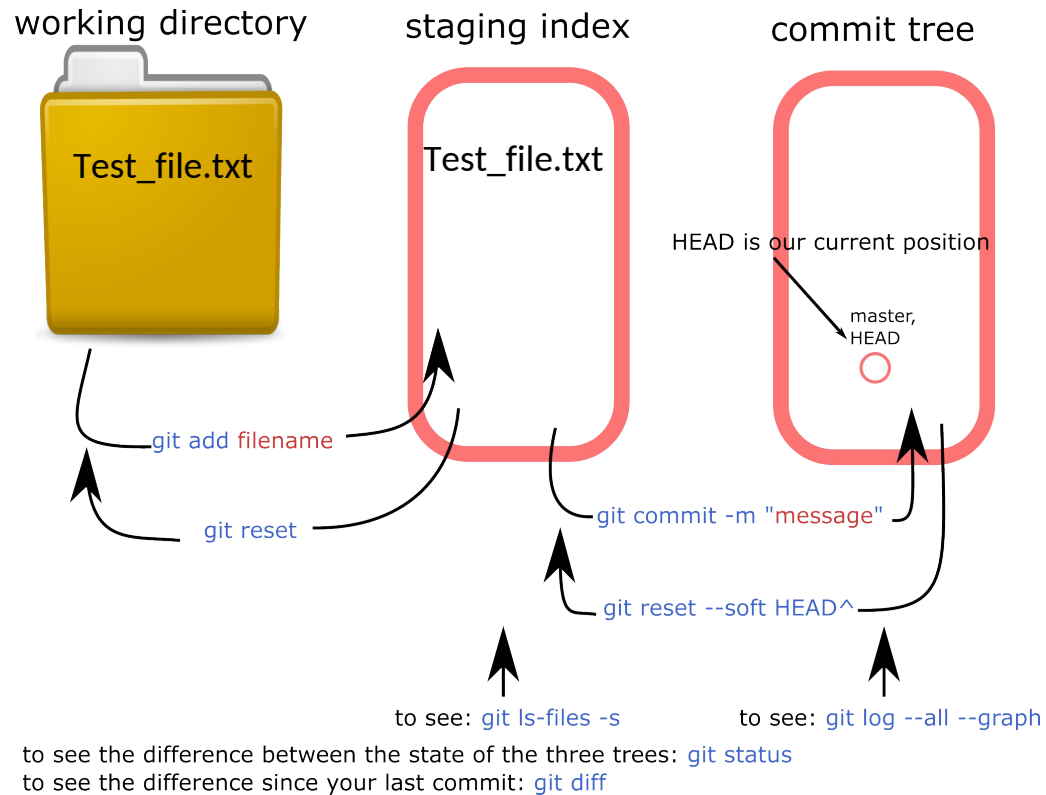
to see the difference between the state of the three trees: `git status`

to see the difference since your last commit: `git diff`

# Git basics: the three trees

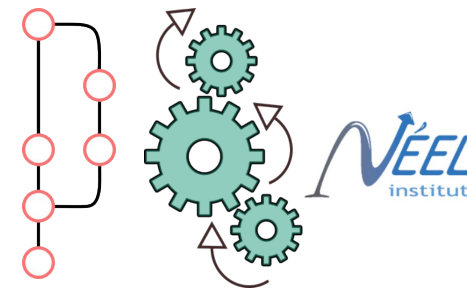


- Now let's put this change in the third tree





# Branches: why and how?

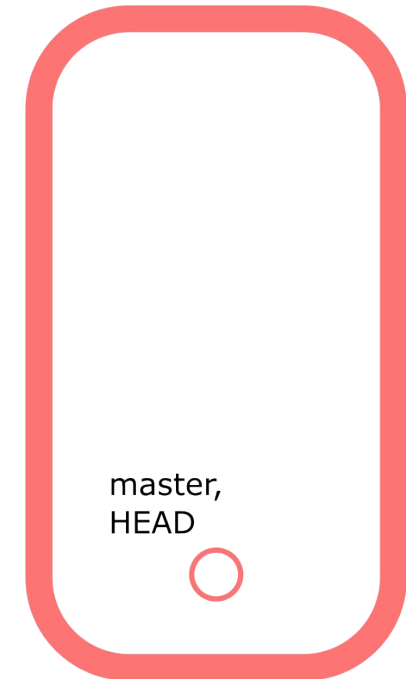


- The branch feature allows to either start developing a feature or access easily some versions of your code
- By default, we are on the branch “master”
- Let us grow a tree:

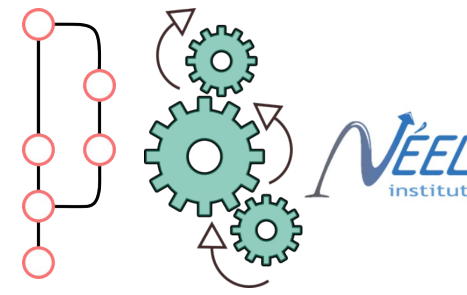
```
~/Documents/Git/test1$ nano function.py
~/Documents/Git/test1$ git add .
~/Documents/Git/test1$ git commit -m "added a function.py with a method to compute a square"
[master 10018df] added a function.py with a method to compute a square
1 file changed, 2 insertions(+)
create mode 100644 function.py
~/Documents/Git/test1$ git log --all --graph
* commit 10018dfba8c6bef8e37c9600b88ac41cca43c0ca (HEAD -> master)
| Author:
| Date: Thu Nov 6 08:37:41 2025 +0100
|
| added a function.py with a method to compute a square
|
* commit 1715f206ef9ba2cb14ba35e3ae97c9f84e1c56ae
Author:
Date: Thu Nov 6 08:31:41 2025 +0100
```

initial commit

commit tree



# Branches: why and how?

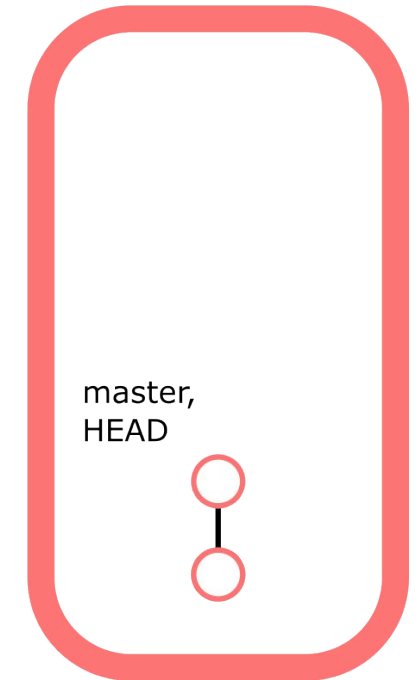


- The branch feature allows to either start developing a feature or access easily some versions of your code
- By default, we are on the branch “master”
- Let us grow a tree:

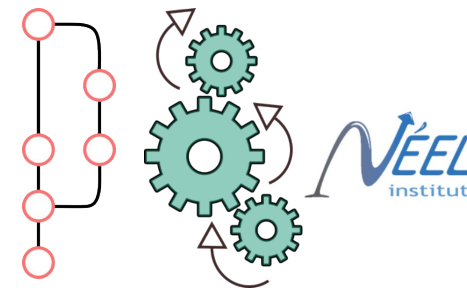
```
~/Documents/Git/test1$ nano function.py
~/Documents/Git/test1$ git add .
~/Documents/Git/test1$ git commit -m "added a function.py with a method to compute a square"
[master 10018df] added a function.py with a method to compute a square
1 file changed, 2 insertions(+)
create mode 100644 function.py
~/Documents/Git/test1$ git log --all --graph
* commit 10018dfba8c6bef8e37c9600b88ac41cca43c0ca (HEAD -> master)
| Author:
| Date: Thu Nov 6 08:37:41 2025 +0100
|
|    added a function.py with a method to compute a square
|
* commit 1715f206ef9ba2cb14ba35e3ae97c9f84e1c56ae
Author:
Date: Thu Nov 6 08:31:41 2025 +0100
```

Here I made some  
changes `def square(x):`  
`return x*x`

commit tree



# Branches: why and how?



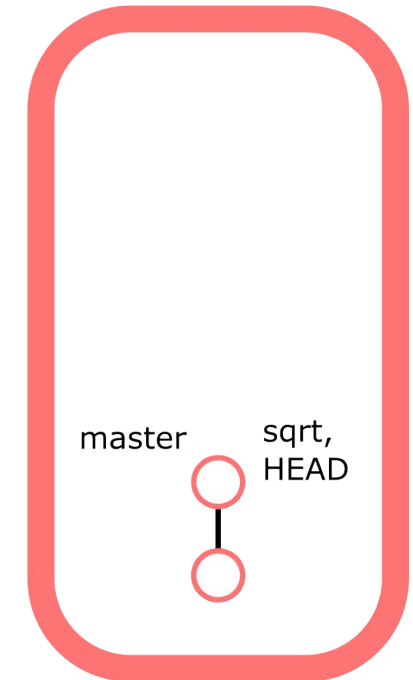
- Now we want to modify the function to return both the square and the square root. Let's make a branch called sqrt:

```
~/Documents/Git/test1$ git branch sqrt  
~/Documents/Git/test1$ git log --all --graph  
* commit 10018dfba8c6bef8e37c9600b88ac41cca43c0ca (HEAD -> master, sqrt)  
| Author:  
| Date: Thu Nov 6 08:37:41 2025 +0100  
| added a function.py with a method to compute a square  
* commit 1715f206ef9ba2cb14ba35e3ae97c9f84e1c56ae  
| Author:  
| Date: Thu Nov 6 08:31:41 2025 +0100  
| initial commit  
~/Documents/Git/test1$ git checkout sqrt  
Switched to branch 'sqrt'  
~/Documents/Git/test1$ git log --all --graph  
* commit 10018dfba8c6bef8e37c9600b88ac41cca43c0ca (HEAD -> sqrt, master)  
| Author:  
| Date: Thu Nov 6 08:37:41 2025 +0100  
| added a function.py with a method to compute a square  
* commit 1715f206ef9ba2cb14ba35e3ae97c9f84e1c56ae  
| Author:  
| Date: Thu Nov 6 08:31:41 2025 +0100  
| initial commit
```

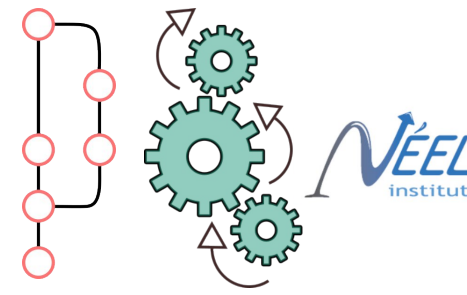
Creates a branch named "sqrt"

Changes the current branch to "sqrt"

commit tree



# Branches: why and how?



- Let us make our modification to it:

Here I made some changes

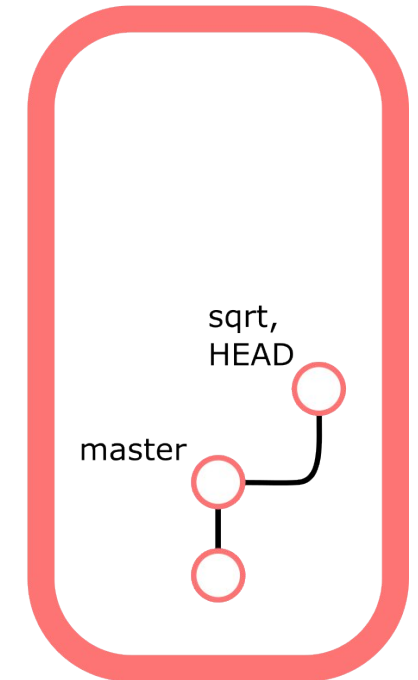
```
def square(x):  
    return x*x
```



```
import numpy as np
```

```
def square(x):  
    return [np.sqrt(x), x*x]
```

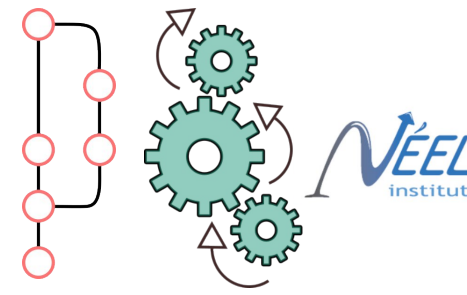
commit tree



```
~/Documents/Git/test1$ nano function.py  
~/Documents/Git/test1$ git add .  
~/Documents/Git/test1$ git commit -m "added the sqrt modification"  
[sqrt 261a8ed] added the sqrt modification  
1 file changed, 3 insertions(+), 1 deletion(-)  
~/Documents/Git/test1$ git log --all --graph  
* commit 261a8ed07e3df94311a230510823d74f5cc74ead (HEAD -> sqrt)  
| Author:  
| Date: Thu Nov 6 09:02:29 2025 +0100  
|  
| added the sqrt modification  
|  
* commit 10018dfba8c6bef8e37c9600b88ac41cca43c0ca (master)  
| Author:  
| Date: Thu Nov 6 08:37:41 2025 +0100  
|  
| added a function.py with a method to compute a square  
|  
* commit 1715f206ef9ba2cb14ba35e3ae97c9f84e1c56ae  
| Author:  
| Date: Thu Nov 6 08:31:41 2025 +0100
```

initial commit

# Branches: why and how?



- Now we are happy with this feature, let us include it in the master

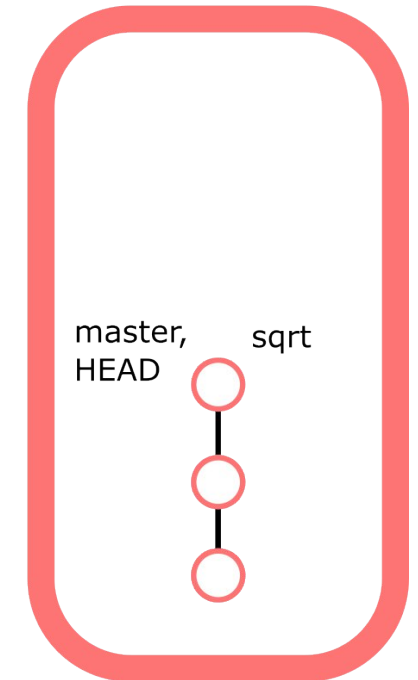
```
~/Documents/Git/test1$ git checkout master
Switched to branch 'master'
~/Documents/Git/test1$ git merge sqrt
Updating 10018df..261a8ed
Fast-forward
 function.py | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
~/Documents/Git/test1$ git log --graph --all
* commit 261a8ed07e3df94311a230510823d74f5cc74ead (HEAD -> master, sqrt)
 | Author:
 | Date: Thu Nov 6 09:02:29 2025 +0100
 |   added the sqrt modification
* commit 10018dfba8c6bef8e37c9600b88ac41cca43c0ca
 | Author:
 | Date: Thu Nov 6 08:37:41 2025 +0100
 |   added a function.py with a method to compute a square
* commit 1715f206ef9ba2cb14ba35e3ae97c9f84e1c56ae
 | Author:
 | Date: Thu Nov 6 08:31:41 2025 +0100
```

Switch back to master

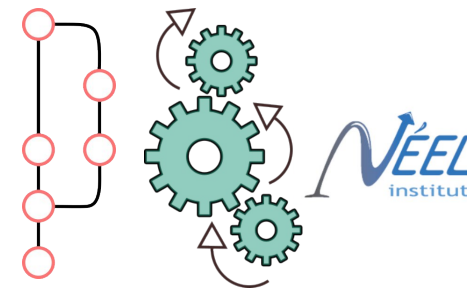
Merge sqrt into master

initial commit

commit tree



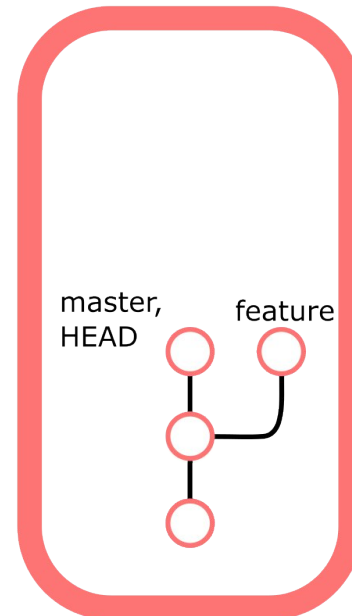
# Conflict solving



- So far, the job was easy: we did not change master while the editing of sqrt happen.
- Let us start from here instead:

What is the outcome of  
“git merge feature” ?

commit tree



File multiply.py

```
def multiply(x,y):  
    return x*y
```

File functions.py

```
def square(x):  
    return x*x
```

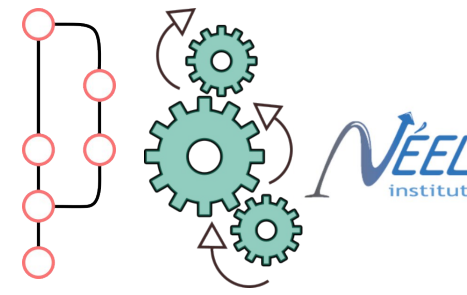
```
def divide(x,y):  
    return x/y
```

File functionbis.py

```
def square(x):  
    return x*x
```

File functions.py

# Conflict solving



- This one was also an easy one: we modified different files in the two branches

commit tree

File functionbis.py

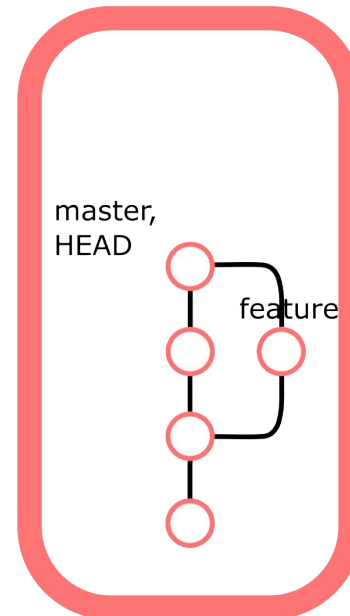
```
def divide(x,y):  
    return x/y
```

File multiply.py

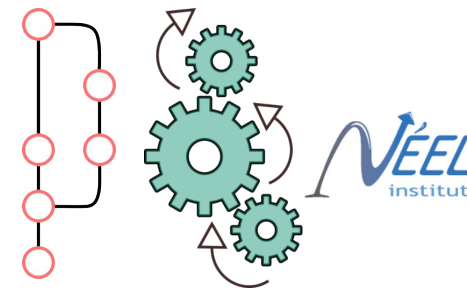
```
def multiply(x,y):  
    return x*y
```

File functions.py

```
def square(x):  
    return x*x
```



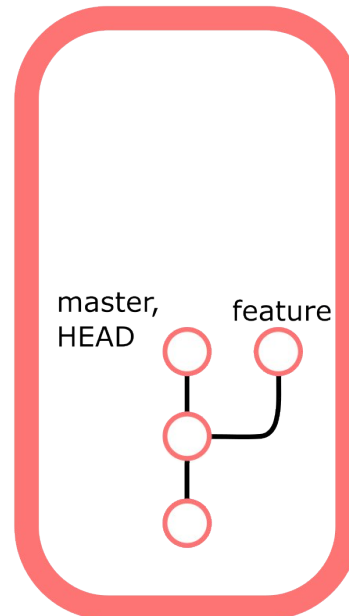
# Conflict solving



- Let us see now what happens in this situation:

What is the outcome of  
“git merge feature” ?

commit tree



File functions.py

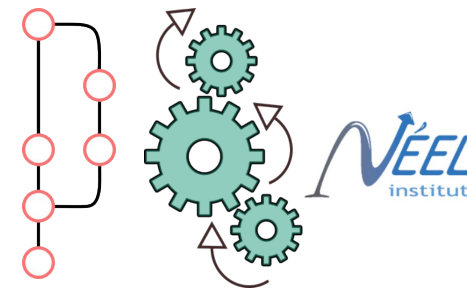
```
def square(x):  
    """ This function returns the square of its input """  
    return x*x
```

File functions.py

```
def square(x):  
    """This function multiplies the input with itself"""  
    return x*x
```



# Conflict solving

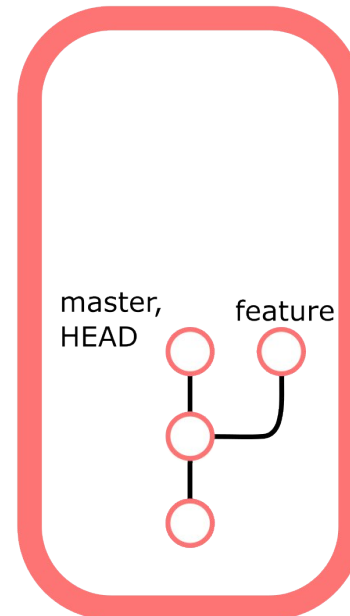


- Let us see now what happens in this situation:

We do not know !

What is the outcome of  
“git merge feature” ?

commit tree



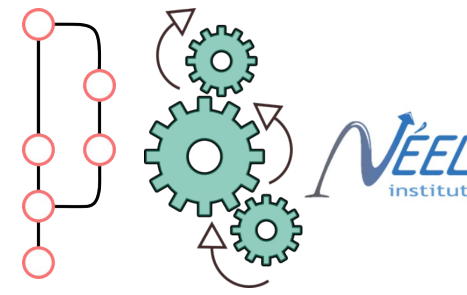
File functions.py

```
def square(x):  
    """ This function returns the square of its input """  
    return x*x
```

File functions.py

```
def square(x):  
    """This function multiplies the input with itself"""  
    return x*x
```

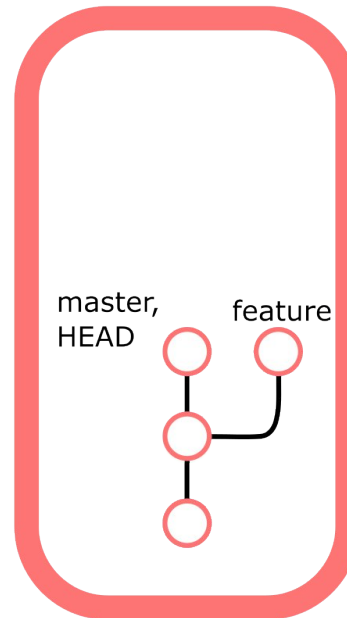
# Conflict solving



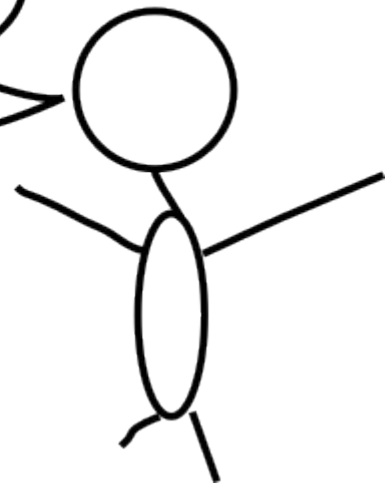
File functions.py

```
def square(x):  
<<<<<<<<<<HEAD  
    """This function returns the square of its input"""  
    =====  
    """This functions multiplies the input with itself"""  
>>>>>>>>>feature  
    return x*x
```

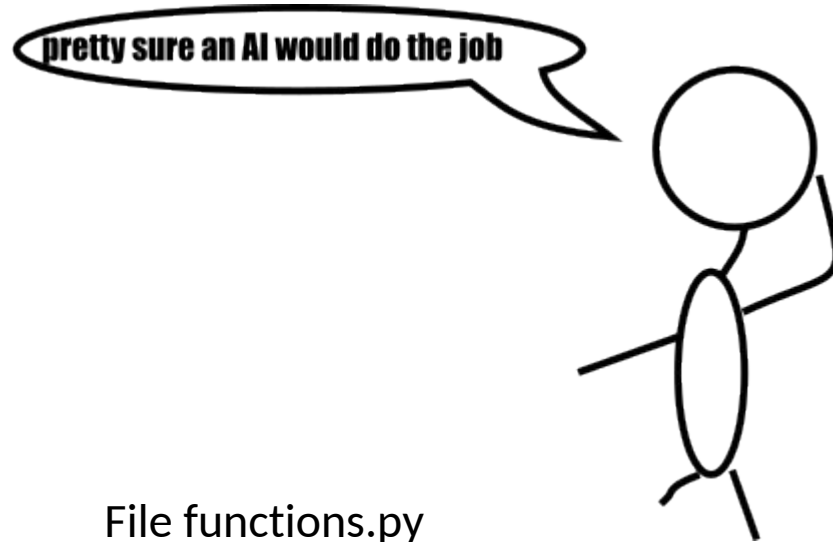
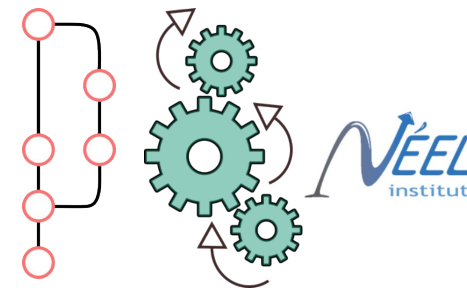
commit tree



**What a mess ! Again a buggy open source software that would have worked just fine if we have just paid a commercial license**



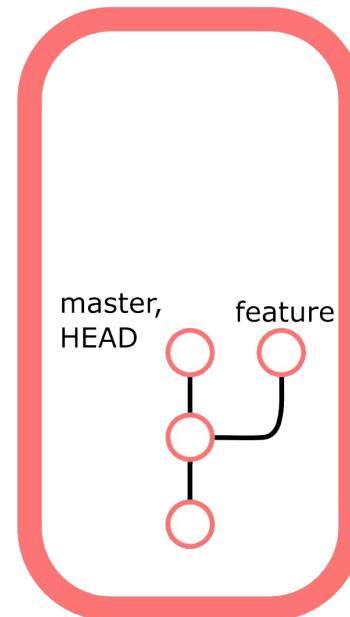
# Conflict solving



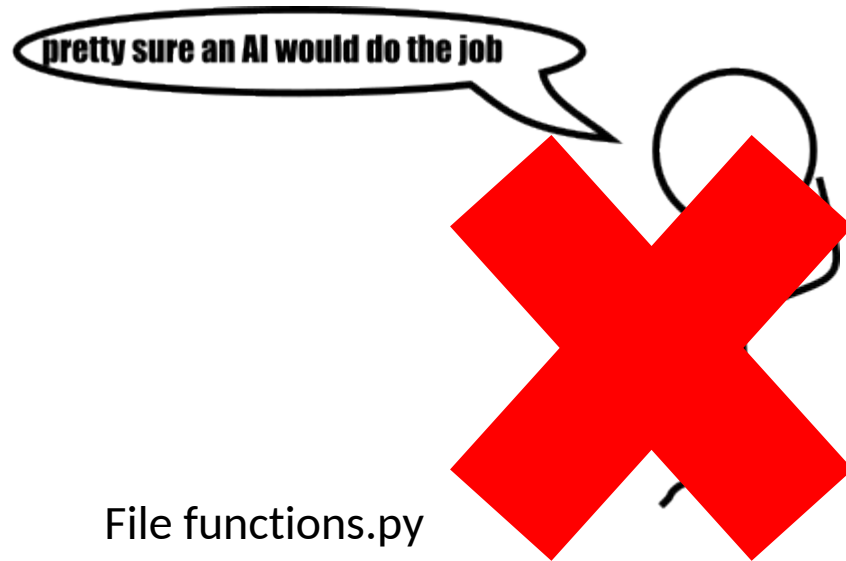
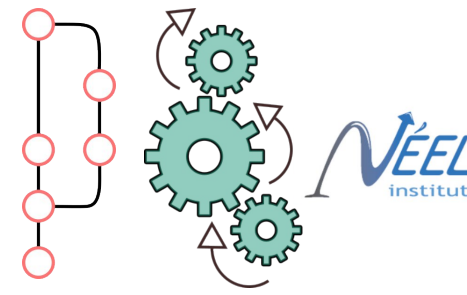
File functions.py

```
def square(x):  
    <<<<<<<<<<<<<HEAD  
        """This function returns the square of its input"""  
    =====  
        """This functions multiplies the input with itself"""  
    >>>>>>>>>>>>>feature  
        return x*x
```

commit tree



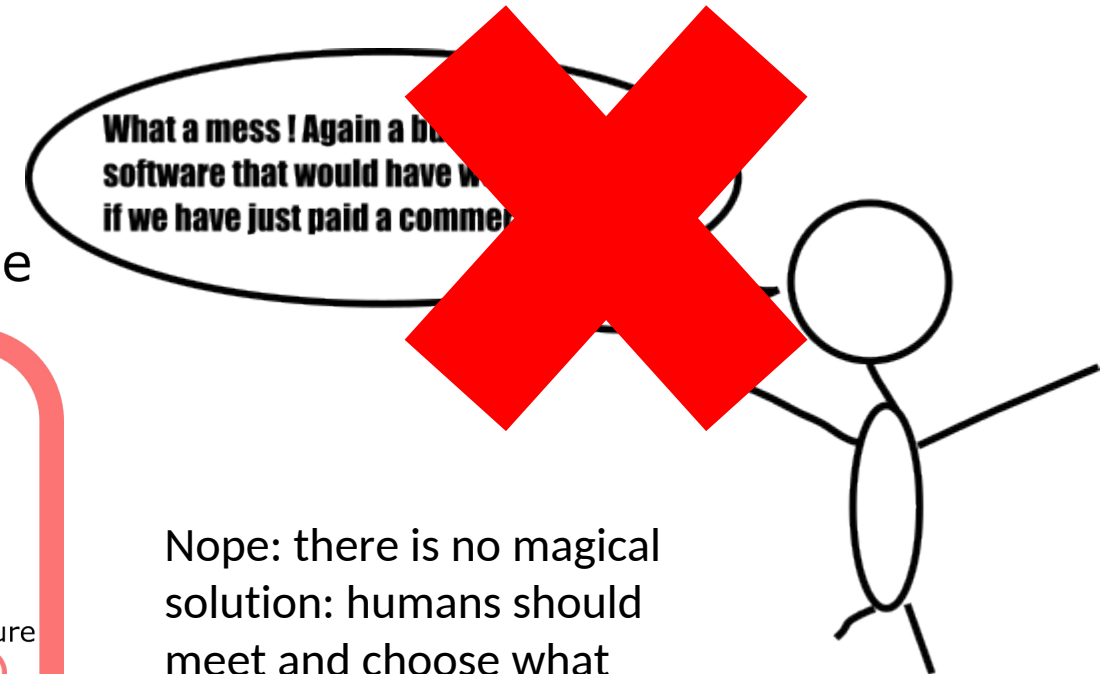
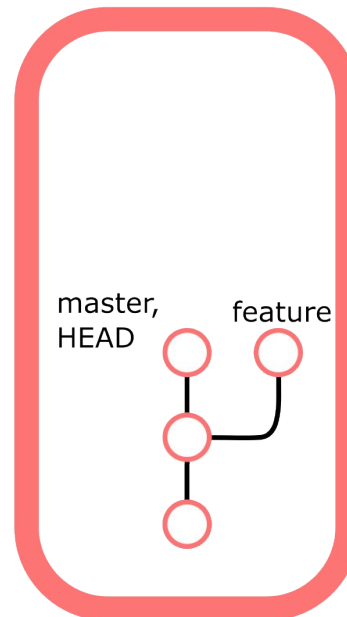
# Conflict solving



File functions.py

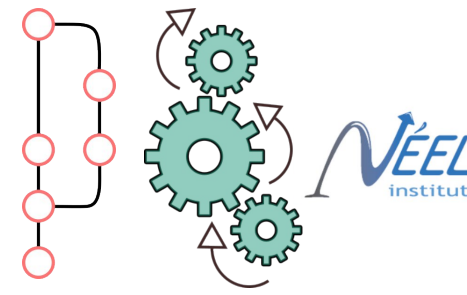
```
def square(x):  
    <<<<<<<<<<HEAD  
    """This function returns the square of its input"""  
    =====  
    """This functions multiplies the input with itself"""  
    >>>>>>>>>feature  
    return x*x
```

commit tree



Nope: there is no magical solution: humans should meet and choose what do they want to keep

# Conflict solving

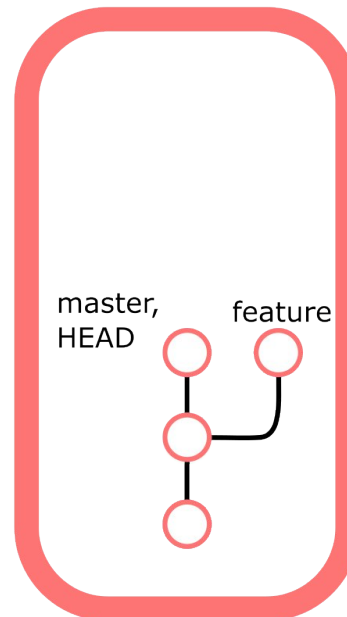


- Git modifies your files to represent a conflict like this:

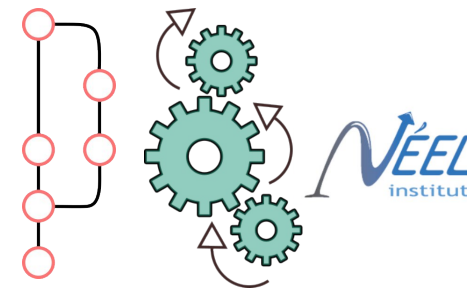
File functions.py

```
def square(x):  
<<<<<<<<<HEAD  
    """This function returns the square of its input"""  
    =====  
    """This functions multiplies the input with itself"""  
>>>>>>>>>feature  
    return x*x
```

commit tree



# Conflict solving



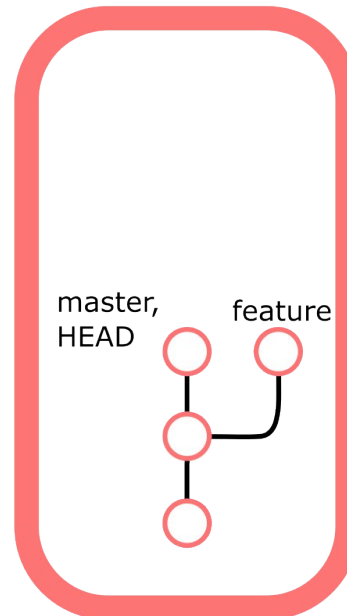
- Git modifies your files to represent a conflict like this:

The modification from master  
(HEAD)

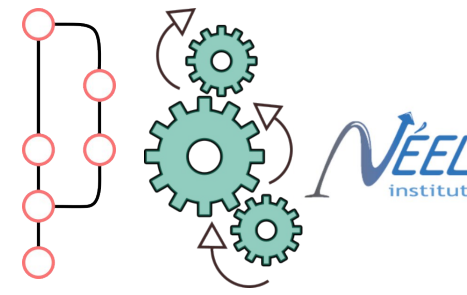
File functions.py

```
def square(x):  
<<<<<<<<<HEAD  
    """This function returns the square of its input"""  
    =====  
    """This functions multiplies the input with itself"""  
>>>>>>>>feature  
    return x*x
```

commit tree



# Conflict solving



- Git modifies your files to represent a conflict like this:

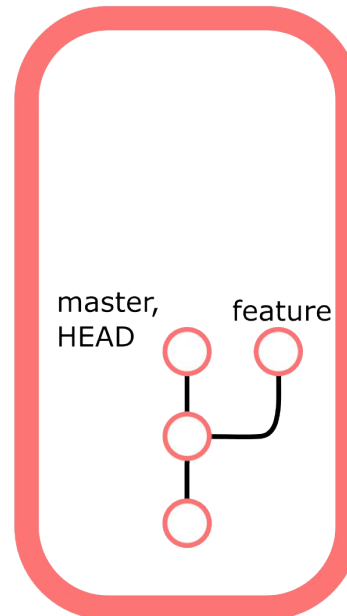
The modification from master  
(HEAD)

File functions.py

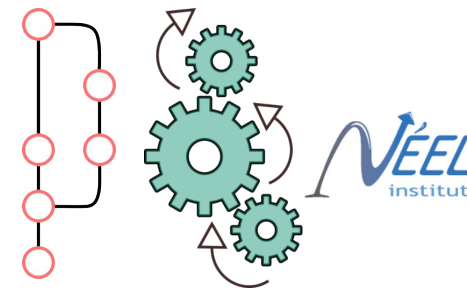
```
def square(x):  
<<<<<<<<<HEAD  
    """This function returns the square of its input"""  
    =====  
    """This functions multiplies the input with itself"""  
>>>>>>>>feature  
    return x*x
```

The modification from feature

commit tree



# Conflict solving



```
~/Documents/Git/git-workshop$ git merge feature
```

Auto-merging functions.py

CONFLICT (content): Merge conflict in functions.py

Automatic merge failed; fix conflicts and then commit the result.

```
~/Documents/Git/git-workshop$ nano functions.py
```

```
~/Documents/Git/git-workshop$ git add functions.py
```

```
~/Documents/Git/git-workshop$ git commit -m "solved the conflict"
```

[master 11a9968] solved the conflict

```
~/Documents/Git/git-workshop$ git status
```

On branch master

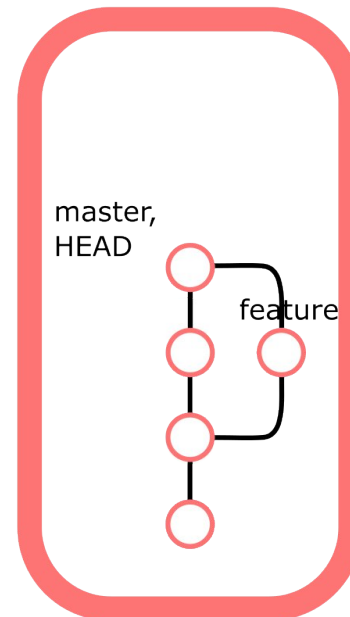
nothing to commit, working tree clean

```
def square(x):
```

```
    """This function gives a little self-hug to the input"""
```

```
    return x*x
```

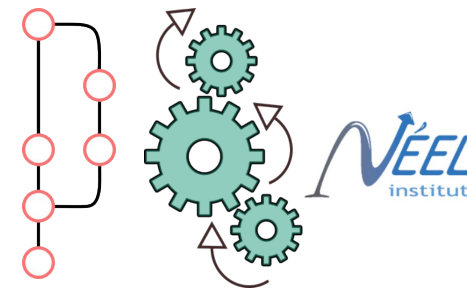
commit tree



You have to edit the file to choose which is the best, or even propose a completely new change



# Conflict solving



```
~/Documents/Git/git-workshop$ git merge feature
```

Auto-merging functions.py

CONFLICT (content): Merge conflict in functions.py

Automatic merge failed; fix conflicts and then commit the result.

```
~/Documents/Git/git-workshop$ nano functions.py
```

```
~/Documents/Git/git-workshop$ git add functions.py
```

```
~/Documents/Git/git-workshop$ git commit -m "solved the conflict"
```

[master 11a9968] solved the conflict

```
~/Documents/Git/git-workshop$ git status
```

On branch master

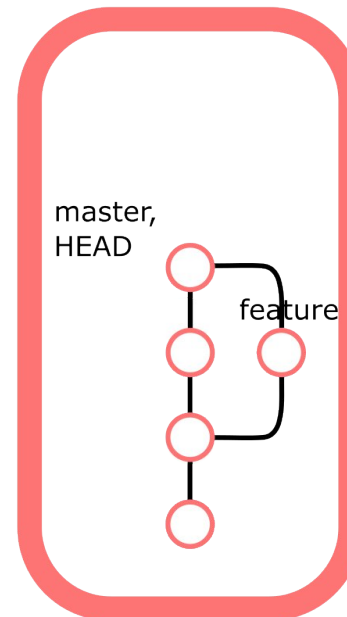
nothing to commit, working tree clean

```
def square(x):
```

```
    """This function gives a little self-hug to the input"""
```

```
    return x*x
```

## commit tree

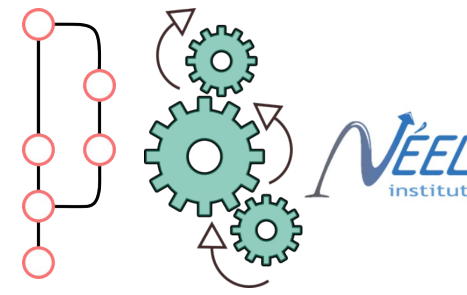


You have to edit the file to choose which is the best, or even propose a completely new change

After merging a branch, we usually delete the old branch, here called feature:

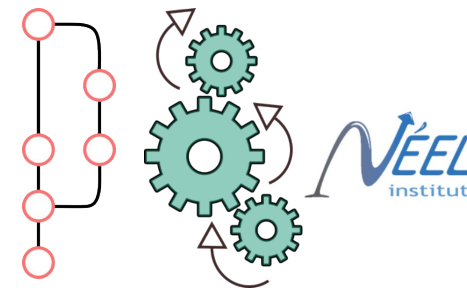
**git branch -D feature**

# Exercise 1 : local branch merging and conflict solving

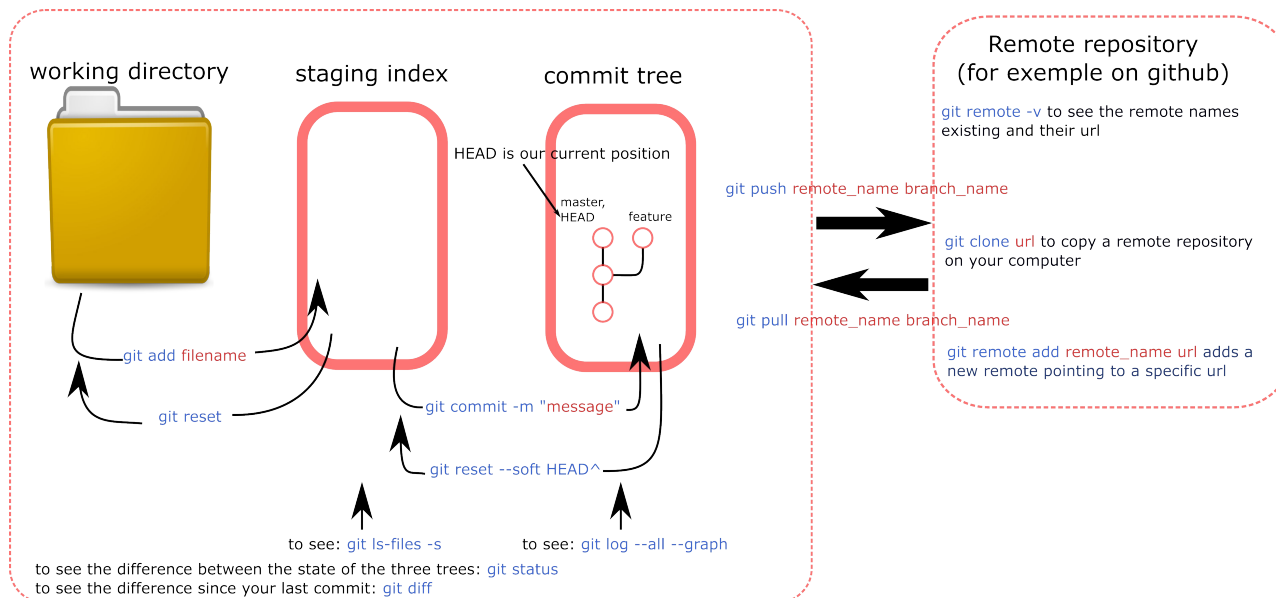



- The task is:
  1. Create a new git repository locally
  2. Add a file, and make a commit
  3. Create two branches, each of them modifying the file
  4. Merge both branches to master, to obtain a conflict
  5. Solve it

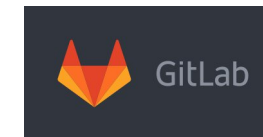
# Remote repositories



- Git is most often used with a remote repository, your local one being one local copy of it.
- It can be stored anywhere, it can be even physically local.

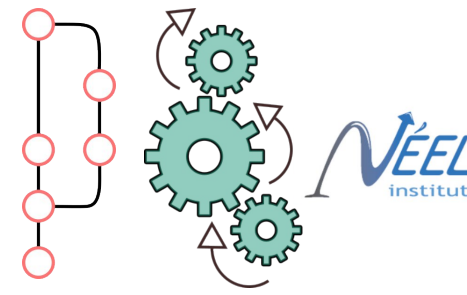


- And there are popular interfaces to host and manage them:
- The most popular is Github  **GitHub**
- But there are other:

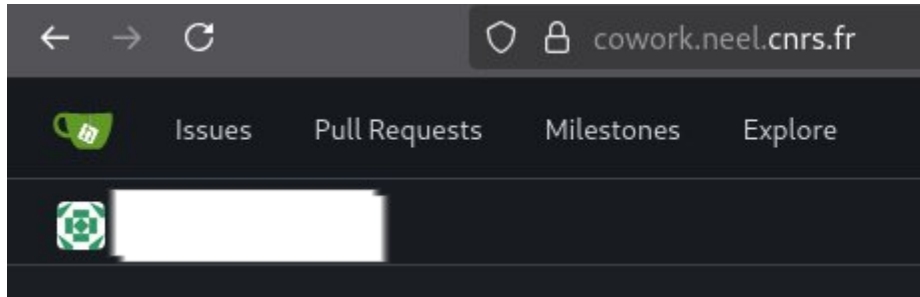


Try for example the NEEL institute git server at [cowork.neel.cnrs.fr](http://cowork.neel.cnrs.fr) with your mail credentials

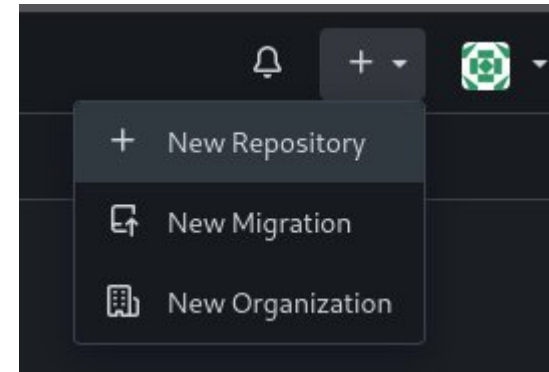
# Remote repositories



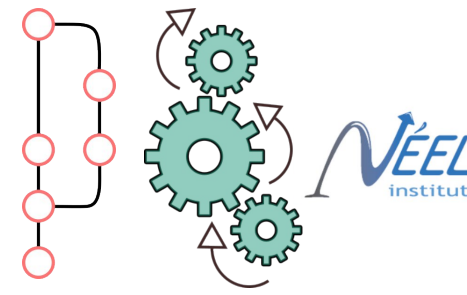
Example with NEEL git  
server



Add a new repository



# Remote repositories




Example with NEEL git server

The only required field is the name



New Repository

A repository contains all project files, including revision history. Already hosting one elsewhere? [Migrate repository.](#)

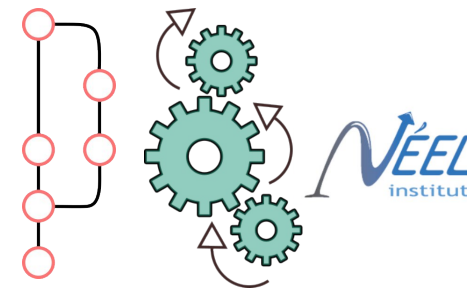
Owner \* 

Some organizations may not show up in the dropdown due to a maximum repository count limit.

Repository Name \*

Good repository names use short, memorable and unique keywords. A repository named ".profile" or ".profile-private" could be used to add a README.md for the user/organization profile.

# Remote repositories

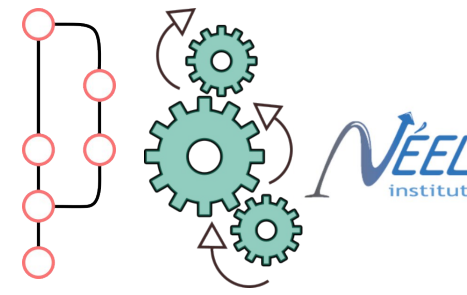


1) choose SSH (you will need to setup ssh keys for this) or HTTPS (a bit less secure but easier to setup)

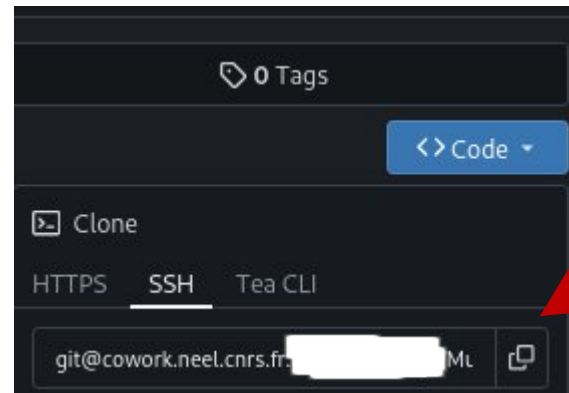
2) depending on the situation, execute in a shell commands close to this or this

A screenshot of a GitHub repository page for a user named [redacted] / git\_workshop. The page has a dark theme. At the top, there are tabs for Code, Issues, Packages, Projects, and Wiki. Below the tabs is a 'Quick Guide' section. The main content area has a heading 'Clone this repository' followed by a link to 'Need help cloning? Visit Help.' Below this are buttons for 'New File', 'Upload File', 'HTTPS', and 'SSH'. The 'SSH' button is highlighted with a red arrow. To the right of the buttons is the repository URL: git@cowork.neel.cnrs.fr:[redacted]:git\_workshop.git. Below the cloning options is a section titled 'Creating a new repository on the command line' which contains a list of shell commands: touch README.md, git init, git checkout -b main, git add README.md, git commit -m "first commit", git remote add origin git@cowork.neel.cnrs.fr:[redacted]:git\_workshop.git, and git push -u origin main. Below this is another section titled 'Pushing an existing repository from the command line' which contains the commands: git remote add origin git@cowork.neel.cnrs.fr:[redacted]:git\_workshop.git and git push -u origin main. A red arrow points from the text 'or this' to the first command in this section. Another red arrow points from the text 'close to this' to the second command in this section.

# Already existing remote repository

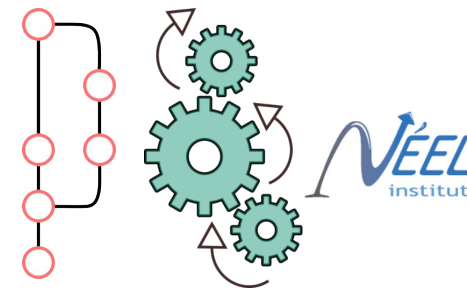


- If the repository already exists, you can make your local copy with the command "**git clone** PATH" where PATH can be found here, after clicking on <code>:



(!) the PATH depends on the authentication method (SSH or HTTPS)

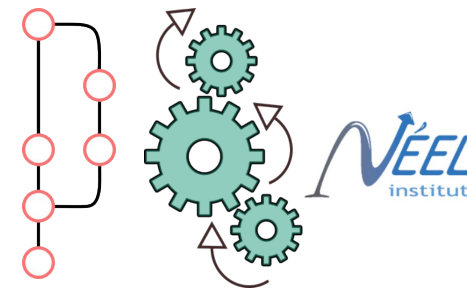
# SSH authentication parenthesis



- We need to set up SSH (there are other way to do the authentication, but this one is the most universal)
- SSH means Secure SHell, it is a protocol to open a shell to talk from a client (in this case, your computer) with a username (in this case, you) to a server (in this case, the Git server [cowork.neel.cnrs.fr](https://cowork.neel.cnrs.fr)) with a server username (in this case, a user called “git”)
- We need to set up RSA keys and if we want a passphrase to do the authentication
- You have an ssh client natively on Linux and Mac. On Windows, we actually installed one when we installed git bash.



# Creating RSA keys



`ssh-keygen -t rsa`

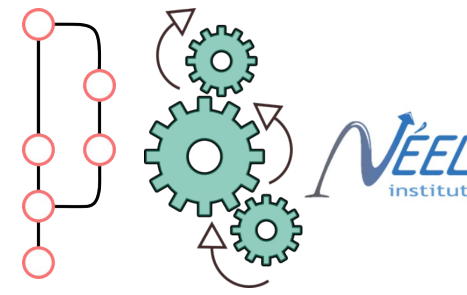


The command to  
execute

It is not mandatory,  
but it is a good idea  
to use a strong  
passphrase when  
asked about it

You will see where are located your private  
key and your public key

# Giving the public key to the server

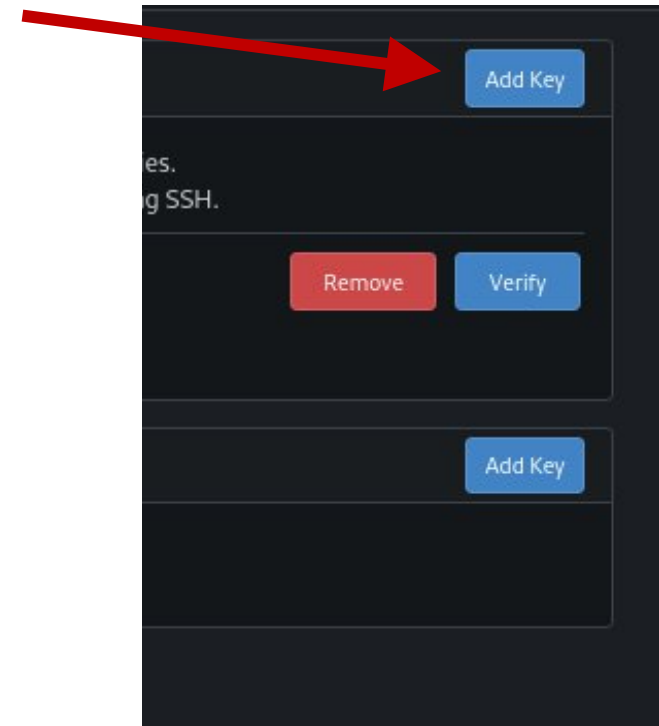
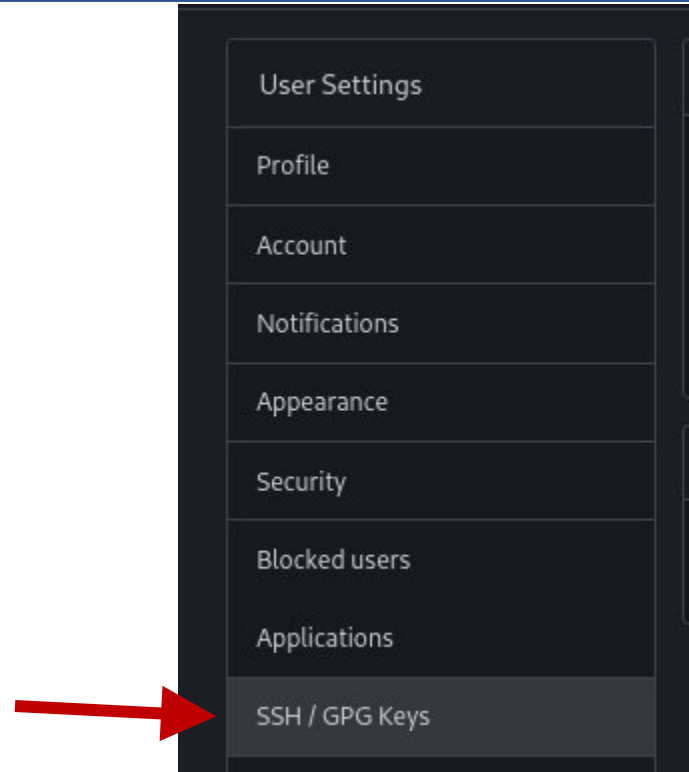
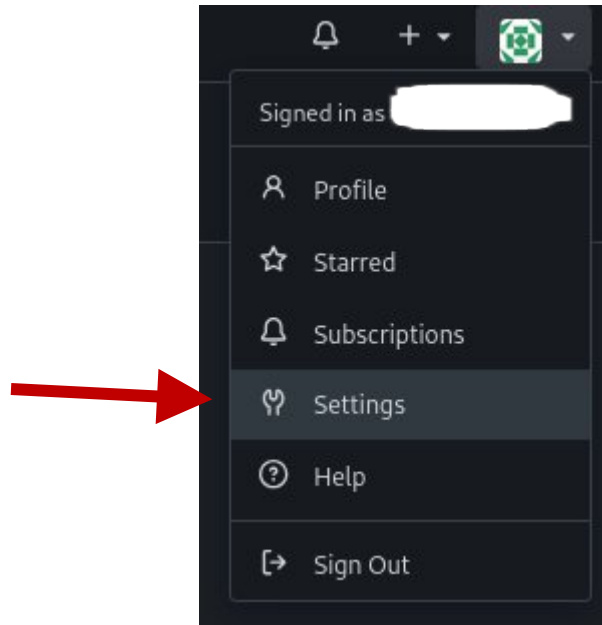
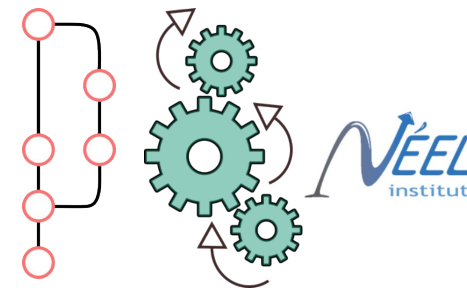


```
cat ~/.ssh/id_rsa.pub
```

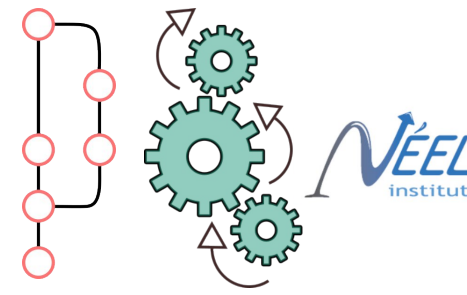
The command to execute to see what is inside your public key

note: you can also open it with a text editor of your choice

# Giving the public key to the server



# Giving the public key to the server

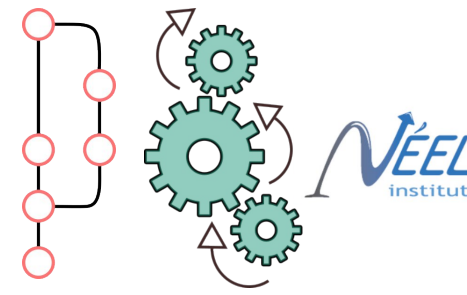
A dark-themed dialog box titled 'Manage SSH Keys' is shown. It has a blue 'Add Key' button in the top right corner. Below the title bar, there is a 'Key Name' label followed by a text input field. Below that is a 'Content' label followed by a larger text area. Inside the text area, there is a hint: 'Begins with 'ssh-ed25519', 'ssh-rsa', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521', 'sk-ecdsa-sha2-nistp256@openssh.com', or 'sk-ssh-ed25519@openssh.com''. At the bottom left of the dialog, there are two buttons: a blue 'Add Key' button and a grey 'Cancel' button. A red arrow points from the 'Add Key' button at the bottom to the 'Add Key' button at the top right.

Give a key name and then paste your PUBLIC key here, then click on "Add Key"

you can then test if you can use this key by typing:

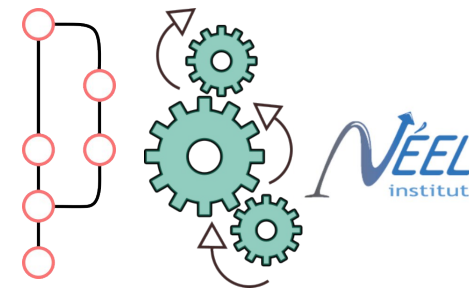
**ssh -T git@cowork.neel.cnrs.fr**

# Note for Github



- For using SSH on Github: same procedure basically
- For using HTTPS on Github: you need to generate a Personal Access Token and use it as your password. This can be done via **Settings/Developer Settings/Personnel Access Tokens/Tokens(classic)**

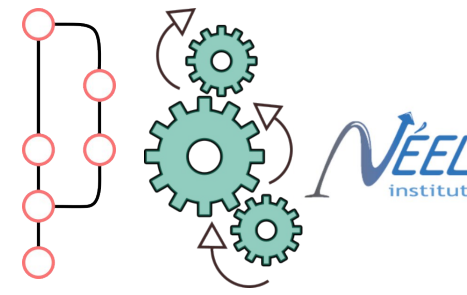
# Pushing the repo



`git push origin master`

the command to use to  
push the branch "master"  
to the remote "origin"

# Add contributors

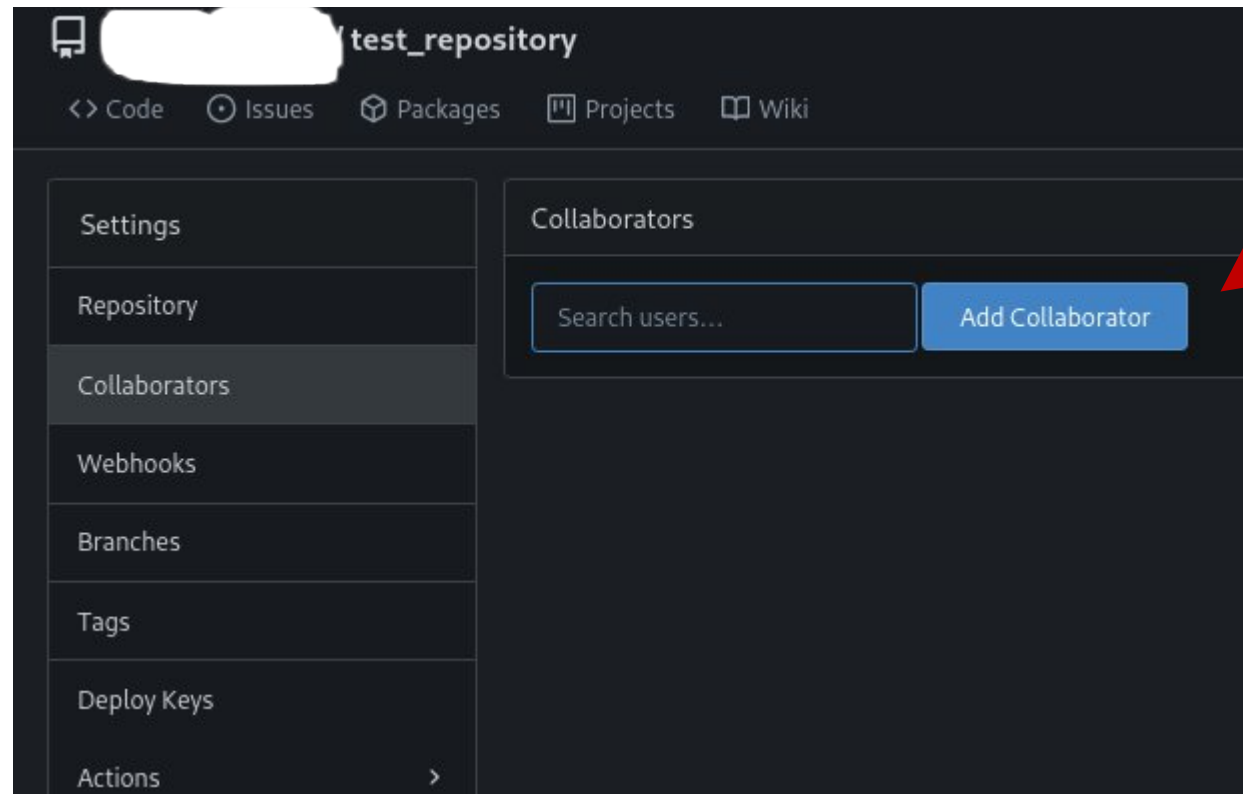
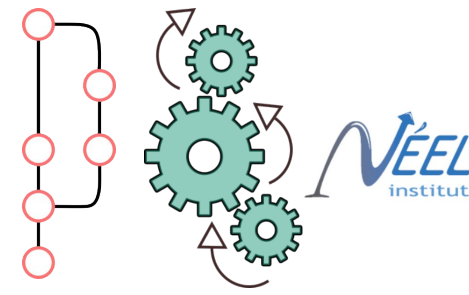
A screenshot of a GitHub repository page for a user named '[redacted]' with the repository name 'git\_workshop'. The page has a dark theme. At the top right, there are buttons for 'Unwatch' (with a count of 1), 'Star' (with a count of 0), and a 'Settings' link. A red arrow points from the 'Unwatch' button towards the top right corner of the image. Below the repository header, there are tabs for 'Code', 'Issues', 'Packages', 'Projects', and 'Wiki'. The 'Code' tab is selected. The main content area includes a 'Quick Guide' section, a 'Clone this repository' section with a link to 'Help', and a 'Creating a new repository on the command line' section. The 'Clone this repository' section shows the repository URL 'git@cowork.neel.cnrs.fr:[redacted]:git\_workshop.git' and buttons for 'New File', 'Upload File', 'HTTPS', and 'SSH'. The 'Creating a new repository on the command line' section contains a code block with the following commands:

```
touch README.md
git init
git checkout -b main
git add README.md
git commit -m "first commit"
git remote add origin git@cowork.neel.cnrs.fr:[redacted]:git_workshop.git
git push -u origin main
```

Below this, there is a 'Pushing an existing repository from the command line' section with a code block containing the following commands:

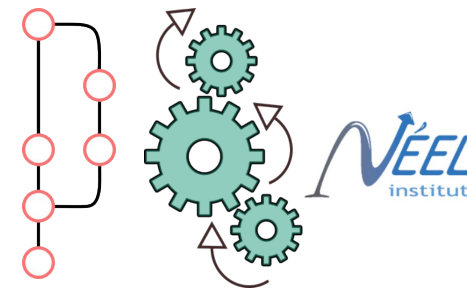
```
git remote add origin git@cowork.neel.cnrs.fr:[redacted]:git_workshop.git
git push -u origin main
```

# Add contributors

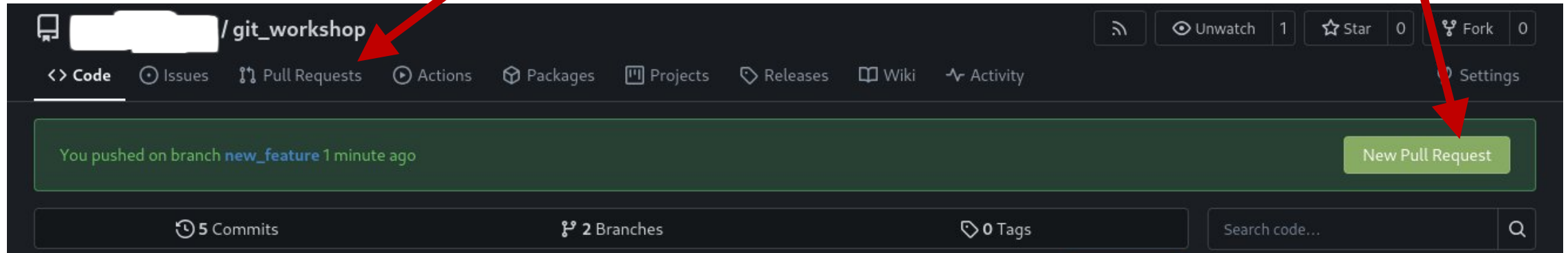




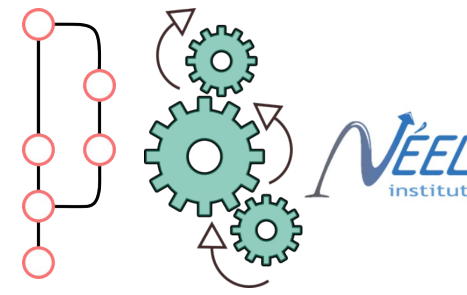
# With this you are ready to do a collaborative work



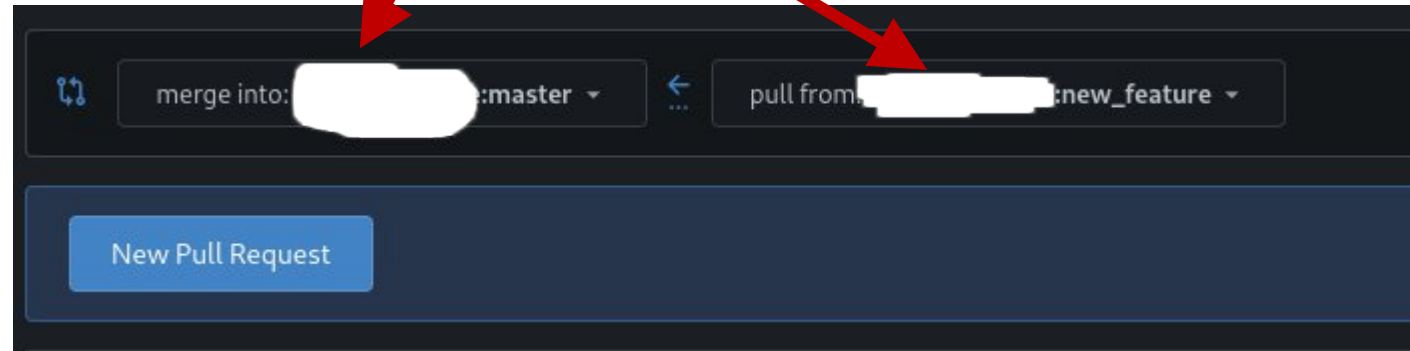
- The cleanest way to merge content collaboratively is called a "pull request":
  1. You make some changes in a branch, for example called "new\_feature"
  2. You push this branch to the remote: "git push origin new\_feature"
  3. You log in the remote server (in your browser: <https://cowork.neel.cnrs.fr>)
  4. You click on "Pull Requests", then "New Pull Request", or directly on "New Pull Request"



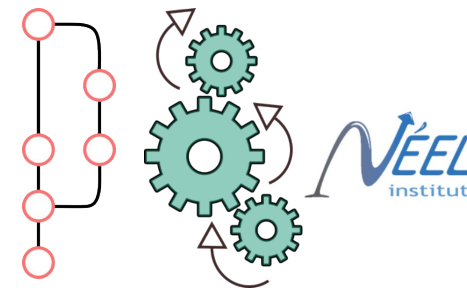
# With this you are ready to do a collaborative work



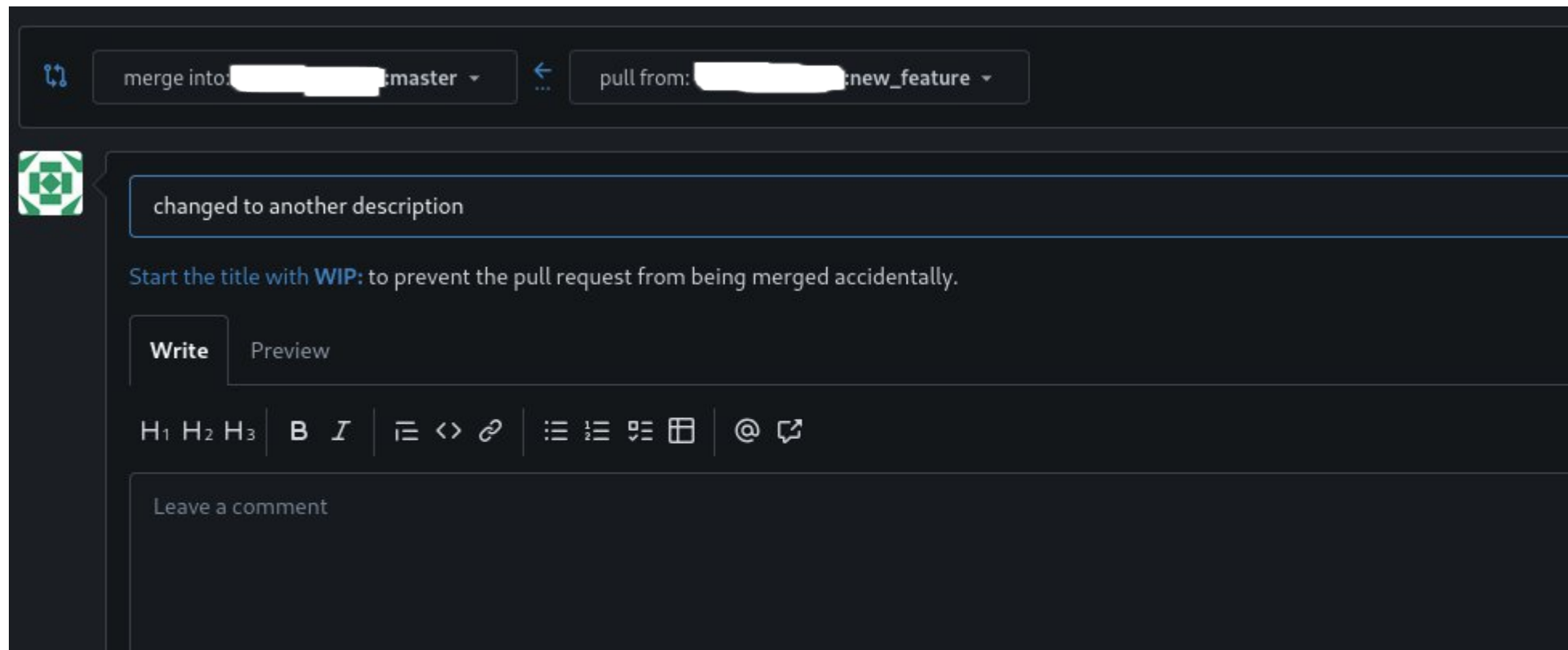
- You select the origin and the destination of the changes



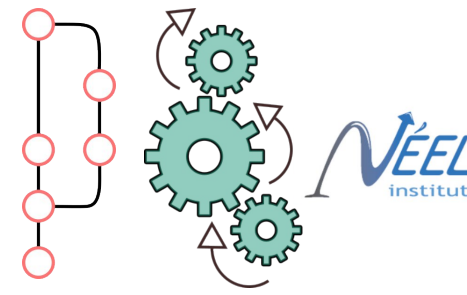
# With this you are ready to do a collaborative work



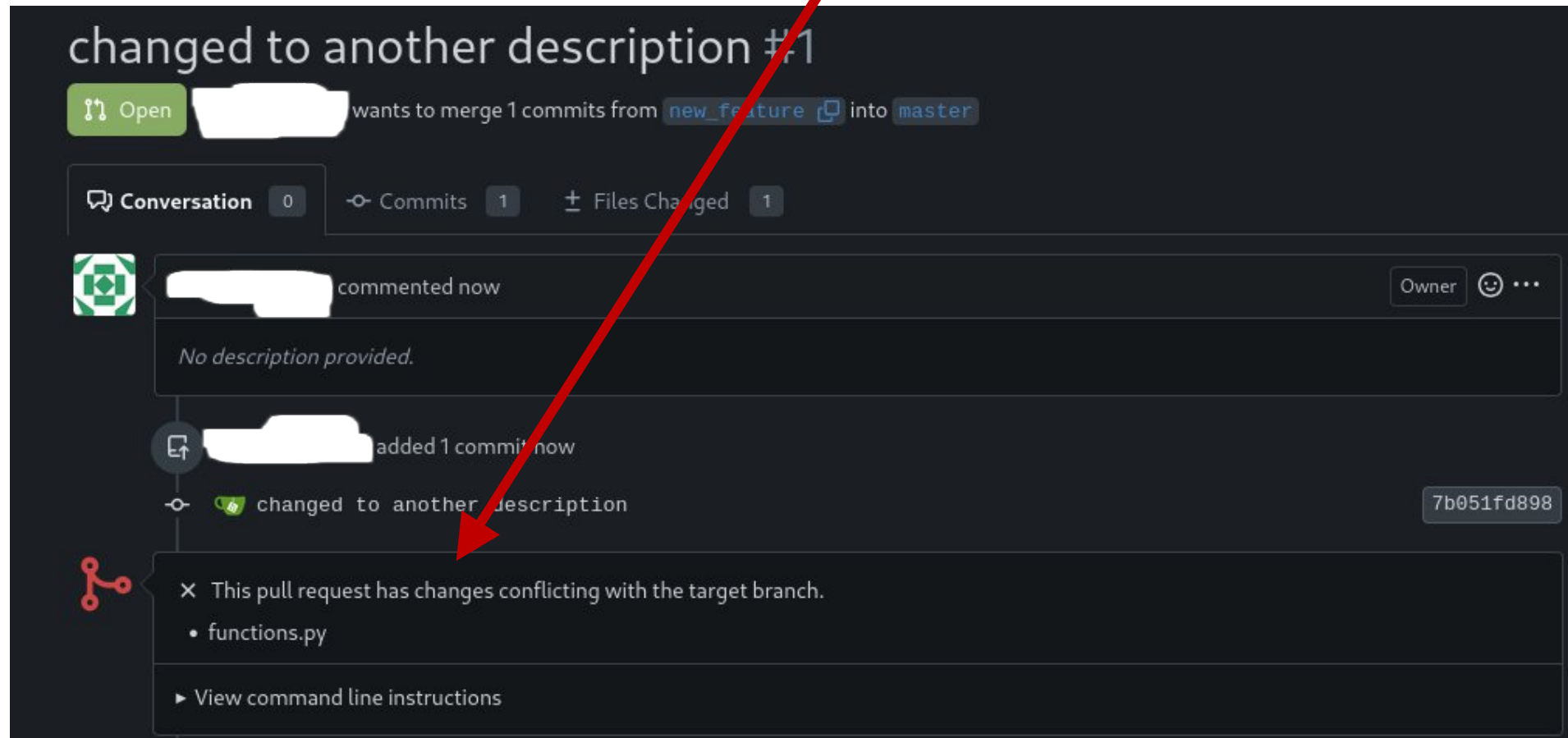
- Put some description of what you want to do, to help your collaborators



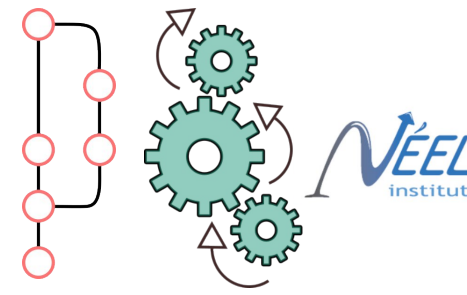
# With this you are ready to do a collaborative work



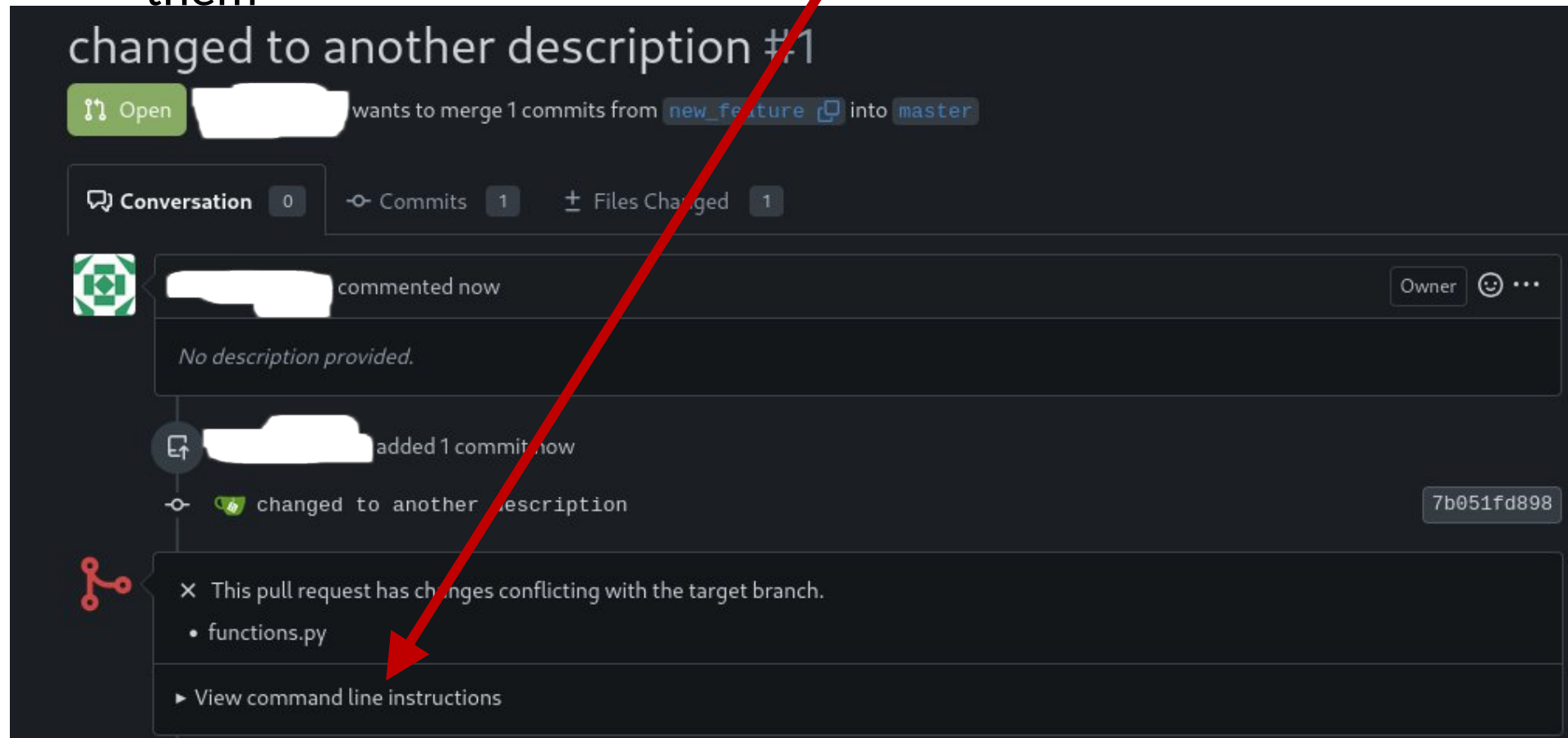
- The GUI will check if your merge will create conflicts



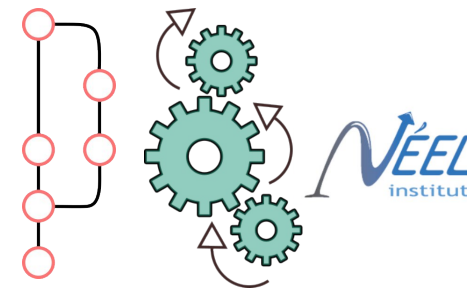
# With this you are ready to do a collaborative work



- The GUI will offer you graphical tools or give you the commands to solve them



# Exercise 2



- Team up two by two: one will be the owner of the remote, the other a contributor
- Create a remote, link it to your local and push something to it
- Add the contributor
- The contributor should « clone » the repo (git clone repo address)
- Both the contributor and the owner will create a branch and propose a conflicting modification, then push it
- You should then merge all those changes in the remote (you can also do everything locally and push the merge results)

# Thank you for your attention

