# Git course

Raoul Grouls

## Table of contents



## Motivation

If you work with code, you will create text files: .txt, .py, .md, .jl, to name just a few. With git, you can make snapshots from the state of your code. The great thing about a snapshot is that you can return to that state again! While simple saving a file might not seem spectacular, consider what happens if you work together on a file. Things get confusion very fast. Git allows you to keep an overview of changes and merge changes in a very exact way.

Sure, you could do this in other ways too. I have seen students that:

- email code
- just "sit together"
- work in different files

But I hope you will agree that none of these are very practical. That is why using git is really the industry standard for collaboration. If you want to work together on code, you will need to learn git.

## First commits

Let's say we make a file `main.py` with one line of code:

```
import numpy as np
```

And now you want to take a snapshot of this. You can do that with:

```
$git add main.py
$git commit -m "main.py import"
```

> **Note**
>
> Putting an `$` before a line means: this is a command you execute in your terminal. You can copy paste it, without the `$`

We have "commited" ourselves to the files we added to the `stage`. The `stage` is where you can gather files before you `commit` them. You can check the stage with:

```
$git status
```

> **Note**
>
> If you want to check what is going on, use `$git status`! You should use it a lot in the beginning, just to understand what is happening.

So now we have a first snapshot with the comment "main.py import" Now, you add a second line.

```
import numpy as np
import pandas as pd
```

If you would run `$git status` you would see something like this:

```
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:   main.py
```

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

As you can read, there is a file modified (`main.py`) and it is not staged. You have a few choices now:

1. You can remove all changes and go back to your last snapshot
2. You can add your changes as a new snapshot.

If you do `$git add main.py` and `$git commit -m 'added pandas'`, you will now have a new snapshot.

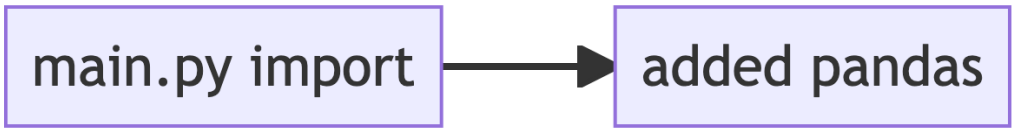


Figure 1: second snapshot

So far, git looks like an unnecessary complex version of saving a file history. But the real strength of git is in what is called branches.

> **Note**
>
> There are a lot of different ways to get an overview of what is going on. I recommend you install the `git graph` plugin in vs code. It gives you a very nice overview of everything that is happening. You could also use the command line. For example, try someting like `$git log --graph --oneline --all` Exit by pressing `q`.
> There are also complete graphical interfaces to git. Especially for complex things, the graph interface is usefull.

## Branching

Now, let's say you and a colleague want to work together on this `main.py` file. You divide the work: one will work on being able to process arguments from the command line, the other will work on the setup of the logger.

By default, git will have a `master` branch (sometimes called `main`). The idea of the `master` branch is, that everything just works there. No bugs. No half features. You develop, experiment and test in branches, and if you are happy with a chunk of working code you merge that into `master`

So, let's branch:

```
$git checkout -b feature/args
```

this will create a new branch, and you will go to that branch immediately (because of the `-b` flag.). Now add to `main.py`:

```
import numpy as np
import pandas as pd
import click

@click.command()
@click.option('--count', default=1)
def main(count)
    print(count)
```

Going back to the old branch is as easy as `$git checkout master`. But first you need to `$git add` and `$git commit` your work. After you did that, you can switch between the two branches. The `feature/args` branch will be one step ahead of the master branch. Like in a tree, the two branches co-exist. They also share a common history, and you can hop between the two branches with the `checkout` command. You can leave out the `-b` flag now, because you dont want to create a new branch, just have a look there.

Now, let's also say that in the mean time, your colleague also worked on this branch. He created a new branch with `$git checkout -b feature/logger` and he added:

```
import numpy as np
import pandas as pd
from loguru import logger
```

After everyone did ther `$git add` and `$git commit`, the state of our project is like this:

There are three branches: (1) master (2) feature/args (3) feature/logger. Let's imagine you are happy with the `feature/args` branch. You can now merge it with:

```
$git checkout master
$git merge feature/args
```

You first jump to master, and there you `merge` the other branch into your master branch. You will get a message that says something like this:

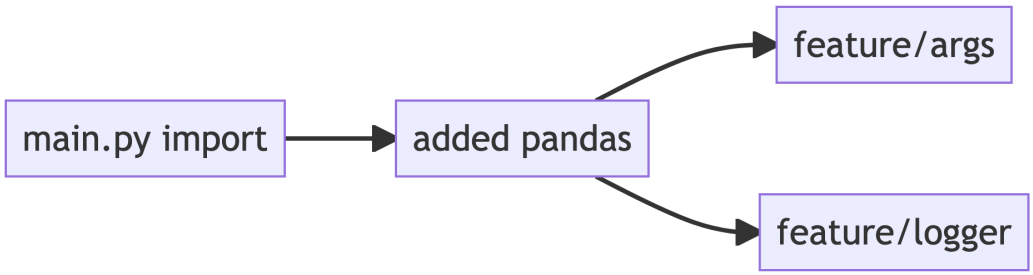


Figure 2: branching

```
 git merge feature/args
Updating a822089..e08a586
Fast-forward
 main.py | 7 +++++++
 1 file changed, 7 insertions(+)
```

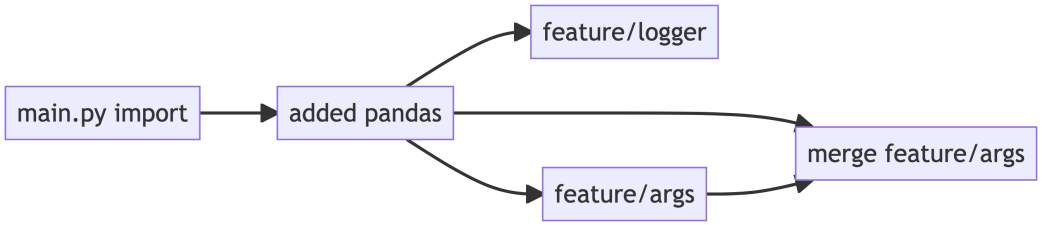


Figure 3: merging

## Conflicts

It might happen that you run into merge conflicts. This happens when you make multiple changes on the same line of code.

In this case, in one file there is an `import click`, and in the other file there is an `from loguru import logger`. You will get a message like this:

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

This is most of the time not a big problem. It just means git wants you to have a look and make a decision.

There are two approaches to this: 1. use a tool to merge. For example, you can use VS code source control. If you click on the file you are trying to merge, it will show the two versions side by side, with highlights on what is changed. With a right click of your mouse you can pick things like 'accept all incoming' or 'accept all current'. 2. edit the code manually. If you want to do more complex things, you can just enter the file with any text editor. You will find parts of the code marked with:

```
<<<<<<< HEAD

=======

<<<<<<< HEAD
```

You can see the code is divided in two parts: a part between «< and ===, and a second part between === and «<. You can simply edit everything the way you want it to look, and remove the «<, === and »> lines. If you use a tool, those lines will be removed by the tool.

After this, you can `$git add` and `$git commit` your fixes, I often do that with a comment like `$git commit -m 'fix merge conflict`.

While it is best to try to avoid merge conflicts, it primarily means you have to look at the code instead of merging automatically.

## Push and pull

To finish things off, instead of emailing your code you can push (send) your git snapshots to a server. Most common are github and gitlab. Setup a repository on github, and you will get instructions how to add the remote repo overthere. After you have added and commited everything, you can just do: `$git push`. You might need to setup your remote branch if you are pushing a new branch for the first time. You will get instructions in the terminal if you need to do that.

Receiving new snapshots in a branch is as easy as: 1. `$git checkout master` (if you want to receive updates from master) 2. `$git pull`

A common error here is that your branch is not clean. This means: you started working in the master branch, and started commiting there. The solution to this is: always checkout to a new branch. Never commit in master. In some organisations, git is set up in a way that it is impossible (for juniors) to commit or push in the master branch. If you made that error: the safest way is to copy the work you want to keep to another branch, and clean out the branch you want to update.How to do that exactly depends a lot on your actual usecase, so I cant write that down here in general.

## Recap

Your workflow should be:

1. edit code
2. git add
3. git commit
4. repeat

Do this often! Some common beginners mistakes are:

a. adding and commiting a LOT of work at once.
b. give commits undescriptive names

You want to avoid a, because it makes it very hard to revert some small part. Let's say you caused an error. With the git system, you can go back to every snapshot and test your code at that point.

You want to avoid b because you want others to understand what you are doing. That "other" can also mean, yourself in three months from now.

So, commit often (everything that is a reasonable unit of functioning work) and describe that unit clearly and to the point.

working together should add to your workflow:

1. `$git checkout master` and `$git pull`
2. branch off from master with `$git checkout -b feature/newfeature`
3. edit, `$git add`, `$git commit`, repeat
4. `$git push`

and, if you are done: 4. `$git checkout master` and `$git merge feature/newfeature`