Coding in Astro
A short introduction to `git`

Sylvain Breton

February 8, 2024

1 Introduction

`git`[1] is a free open source tool dedicated to code version control. It is almost universally used in modern software development. It is unavoidable when working on large projects involving several developer while being a great asset to keep track on the changes and evolution you apply on your personal projects. To go through the tutorial presented in this document, you should first create an account on GitLab or GitHub (having an account on both platforms is always useful).

From experience, I would say that `git` is a very good example of *learn by practice* tool. Some concepts that appear somewhat abstract or useless for the beginner's eye reveal their relevance only after a few times of struggling with code management and thinking *Ok things would be so much simpler if I were able to do this !* I would therefore advise you to force yourself to use `git` on each of your new project, and you will see that you steadily progress in skill and understanding.

Finally, as disclaimer, note that this guide is not meant to provide an exhaustive description of `git` possibilities, but rather to walk you throughout some key functionalities and allow you to explore what remain by yourself. Have fun !

2 Installation

Before starting the tutorial, you should check that `git` is installed in your system. Just type in the console

```
$ git
```

and if the command is unknown, follow the installation instructions from the official documentation: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git.

3 Your first repository

3.1 `git` configuration

Before proceeding to the rest of the tutorial, you should be sure that `git` knows at least your username and your email address. You can check which elements are already configured by doing

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

If they are not, do

```
$ git config --global user.name "Roger of Sicily"
$ git config --global user.email roger_of_sicily@example.com
```

Additional configuration options are described here on the official documentation: https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration. In particular, it can be useful to configure the name of the repository default branch to `main`

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

[1] See `git` official documentation: https://git-scm.com.

3.2 Creating the repository

Open the console and navigate to the point where you want to create the repository

```
$ mkdir tutorial
$ cd tutorial
```

Once you are there, create a file that you will name `README.md`. Inside the `README.md` file, write something like[2]

```
# My first repository

Hello world ! This is my first repository !
```

You will note that until now we have done nothing with `git`. Let our directory become a repository !

```
$ git init
$ git add README.md
$ git commit -m "Adding README file"
```

The `init` command initiated the repository. The `add` command added `README.md` as a `git` tracked file and included it to be considered in the next commit. The files to be included in the commit, the files with change but not yet included, and the untracked files, can be displayed by doing

```
$ git status
```

The `commit` command add the change just made to the history log of the repo, that you can display with

```
$ git log
```

3.3 Creating the distant repository

One of the main asset of using `git` is to manage your local repositories together with distant repositories hosted on platforms such as `GitLab` or `GitHub`. In what follows, I will consider `GitLab` for the example that is of interest for us. With your `GitLab` account, click on *New project*. Call it *tutorial* and make sure you untick the sentence asking if you want to initialise your new repository with a `README.md` file. Once it is created, follow the three last lines of the instructions provided in the *Push an existing Git repository* case

```
$ git remote add origin git@gitlab.com:sybreton/tutorial.git
$ git push --set-upstream origin --all
$ git push --set-upstream origin --tags
```

3.4 Cloning an existing repository

You will very often work with distant repository collected from `GitLab` or `GitHub`. You can clone either using HTTPS (first syntax) or SSH (second syntax):

```
$ git clone git@gitlab.com:the_repo_location
$ git clone https://@gitlab.com:the_repo_location
```

3.5 Some standard repository structure

The internal tree structure of most `Python` package is as follows

```
tutorial
├── README.md
├── MANIFEST.in
├── requirements.txt
├── LICENSE
├── pyproject.toml
└── setup.py
```

[2] This is a Markdown file, that you can format using the syntax instructions described here: https://www.markdownguide.org/basic-syntax/.

```
├── docs
├── src
│   └── tutorial
│       ├── __init__.py
│       └── source_code.py
├── tests
│   └── some_test_file.py
├── notebooks
│   └── a_tutorial_notebook.ipynb
└── .gitignore
```

Although it would be too long for this tutorial to describe the exact interest of all these files and repo, you can note that the actual source code will be put in the `src/tutorial` directory. To continue with the following sections of the tutorial, create at list the `src/tutorial` directory.

3.6 Branches

A key feature of `git` is the possibility to have, on one repository, different branches with each of them displaying the code you are working on in diverse states. You can show the existing branches with

```
$ git branch
```

In particular, it displays which the active branch is. The current standard name of the reference branch is `main`. It can be useful to create a new branch, let's call it `dev`.

```
$ git branch dev
```

You can switch branches by doing

```
$ git checkout dev
```

Note that you can create a branch and instantaneously make it the active branch

```
$ git checkout -b dev
```

Now, go to `src/tutorial` and create the `source_code.py` file with inside

```
def my_function () :
    """
    A useless function that prints ``Hello world``.
    """
    print ("Hello world !")
```

We are going to add the file, commit the change, and push to new distant branch that our local `dev` will now be tracking

```
$ git add source_code.py
$ git commit -m "Add file with crucial source code"
$ git push -u origin dev
```

Note that, once your local branch knows which branch to track, you can simply push on the distant repo by doing

```
$ git push
```

As `dev` is now different from `main`, you can compare the two branches by doing (assuming your active branch is still `dev`)

```
$ git diff main
```

If your local branch is lacking commits with respect to the distant branch, you will need to do

```
$ git pull
```

Finally, note that you can fetch a distant branch with the following instruction

```
$ git fetch origin dev
```

This functionality might prove useful when you are doing advanced operations such as *rebasing* (see Sect. 4.2).

4 More advanced functionalities

4.1 Merging

You probably wonder how to apply the development you made on `dev` onto `main`. This operation is called *merging*.

```
$ git checkout main
$ git merge dev
```

In case the two branches have conflicting histories, `git` will highlight the conflict between branches, ask you to make the required modification, then commit them. In our case, a simple *fast-forward* is sufficient here.

4.2 Diverging commit logs: rebasing

Rebasing is maybe the most important of the advanced features provided by `git`. The core idea is to allow to branch that diverged in terms of commit history to share again the same linear commit history, by putting all commits of one given branch behind the ones of the other[3]. Mastering this functionality is key when different person work on different branches/fork (see below what *forking* is) of the same project and want to keep a clean project history.

To illustrate this functionality, change the function you created to

```
def my_function () :
    """
    A useless function that prints ‘‘Hello world‘‘.
    """
    print ("Hello world (main branch version) !")
```

then commit the change and checkout to dev (which will not own this modification yet).

```
$ git commit source_code.py -m "Main branch version"
$ git checkout dev
```

On `dev`, make the following change

```
def my_function () :
    """
    A useless function that prints ‘‘Hello world‘‘.
    """
    print ("Hello world (dev branch version) !")
```

then commit it

```
$ git commit source_code.py -m "Dev branch version"
```

Now, let's ask for the rebase operation.

```
$ git rebase main
```

There are merge conflicts, and `git` will mention it with a detailed statement !

```
First, rewinding head to replay your work on top of it...
Applying: Dev branch version
Using index info to reconstruct a base tree...
M src/tutorial/source_code.py
Falling back to patching base and 3-way merge...
Auto-merging src/tutorial/source_code.py
CONFLICT (content): Merge conflict in src/tutorial/source_code.py
error: Failed to merge in the changes.
Patch failed at 0001 Dev branch version
hint: Use 'git am --show-current-patch' to see the failed patch
Resolve all conflicts manually, mark them as resolved with
```

[3]Obviously, you could just merge the two branches as previously shown, but you would still have to manually fix the branch conflict and that would produce an ugly non-linear commit log.

```
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
```

Here is what you should get inside `source_code.py`, with the location of the merge conflicts highlighted by `git`

```
def my_function () :
    """
    A useless function that prints ‘‘Hello world‘‘.
    """
<<<<<<< HEAD
    print ("Hello world (main branch version) !")
=======
    print ("Hello world (dev branch version) !")
>>>>>>> Dev branch version
```

Edit the file to have again the code version you wanted applied at this step, that is

```
    print ("Hello world (dev branch version) !")
```

then proceed with the instruction displayed in the console, that is

```
    $ git add source_code.py
    $ git rebase --continue
```

You now have a clean commit history if you do `git log` !

4.3 Tagging

You might want to put easy to retrieve labels on keystone version of your project. Tags are here for this !

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

To display existing tags, just do

```
    $ git tag
```

You can look at a specific tag information by doing

```
    $ git show v1.0
```

They should be pushed explicitly on distant repositories

```
    $ git push --tags
```

Note that you can navigate from one commit to another, or one tag to another, using the `checkout` command.

```
    $ git checkout v1.0
```

4.4 `.gitignore` files

Doing `git status` also display the list of untracked files. Sometimes, there can be a lot of them and this might be a problem for readability of the command output. To prevent this, you can create a `.gitignore` file at the root of your project in order to specify file patterns that should not be tracked. Note that `GitLab` and `GitHub` can automatically create for you a `.gitignore` template with file patterns to ignore selected regarding the language you are using for your project. You can read more about `.gitignore` on the online documentation: https://git-scm.com/docs/gitignore.

4.5 Exchanging SSH keys

If you pull or push commits from a private distant repository, `GitHub` or `GitLab` will require you to enter your username and password each time you need to interact with the distant repository, which might prove a little bit fastidious. The way around this is to set up a SSH key on your computer and to share its public signature with the distant platform. Below are the links with more detailed explanation both for `GitLab` and `GitHub`:

- https://docs.gitlab.com/ee/user/ssh.html
- https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account

4.6 Forking

Both `GitLab` and `GitHub` offer the possibility to create your own repository from a repository owned by someone else. This is called *forking*, and will allow you to work on your version of the repository as maintainer. The two repositories remain connected in the sense that you can ask for *merge requests* from one of your branches to the original repository to provide some developments you want to share.

4.7 Pipelines

When your project reach an advanced state and that you have a ready-to-deploy module, it is useful to know that `GitLab` and `GitHub` allow you to use automated testing and deployment pipeline (Continuous Integration / Continuous Delivery, CI/CD) on the platform[4]. The action to execute are provided at the root of the repository, in a file named `.gitlab-ci.yml`, in the case of `GitLab`. Below is a simple example of CI/CD pipeline that simply check, in two stages, that the module is able to correctly install and that the test run properly:

```
stages:
 - build
 - test

build:
 stage: build
 image: python:3.9
 script:
  - pip install -r requirements.txt
  - pip install .

test:
 stage: test
 image: python:3.9
 before_script:
  - pip install -r requirements.txt
  - pip install .
  - pip install pytest==7.2.0
 script:
  - pytest tests/some_test_file.py
```

[4] You can read more about here e.g. in the case of `GitLab`: https://docs.gitlab.com/ee/ci/