Introduction to Git

IN104: Projet Informatique[2]

Natalia Díaz Rodríguez

ENSTA ParisTech

March 2019

[2]♡Acknowledgment: Slides extended from Florence Carton, Antonin Raffin & Ugo Vollhardt

Table of contents

What are Version Control Systems (VCS)?

- A VCS tracks the history of changes as people and teams collaborate on projects together.
- As the project evolves, teams run tests, fix bugs, and contribute new code
  - with the confidence that any version can be recovered at any time.
- Developers can review project history to find
  - Which changes were made?
  - Who made the changes?
  - When were the changes made?
  - Why were changes needed?

Distributed Version Control Systems (DVCS)

- Git: an example of a DVCS commonly used for open source and commercial software development.
- DVCSs allow full access to
  - Every file, branch, and iteration of a project
  - A history of all changes.
- Git and other VCSs:
  - Help team members stay aligned through a unified and consistent view of the project while working independently.
  - Don't need a constant connection to a central repository: Developers can work anywhere and collaborate asynchronously from any time zone.
- Without version control, team members are subject to:
  - Redundant tasks
  - Slower timeline
  - Multiple copies of a single project.

Git

Many revision control systems: **Why Git?**

- Need a place to store code when team size +1
- Git has over 10M repos
- Github offers free private repos (now for everyone!)
- Allows every developer to work on the same file (and have a local copy)

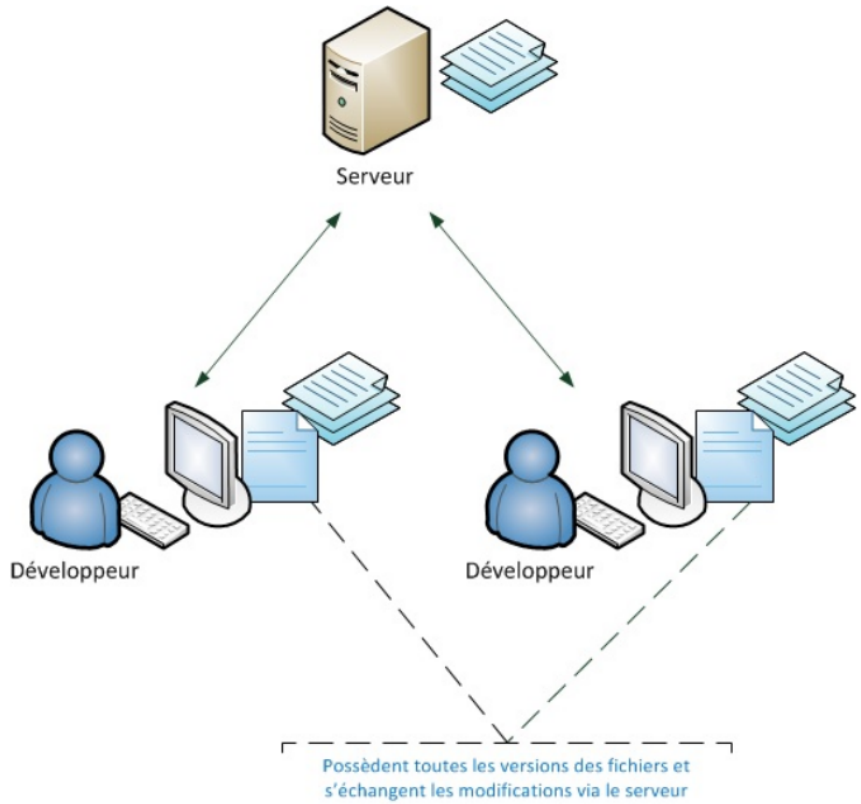


Figure: Git[a]

[a] www.openclassrooms.com/courses/gerer-son-code-avec-git-et-gith

Git

Why Git?

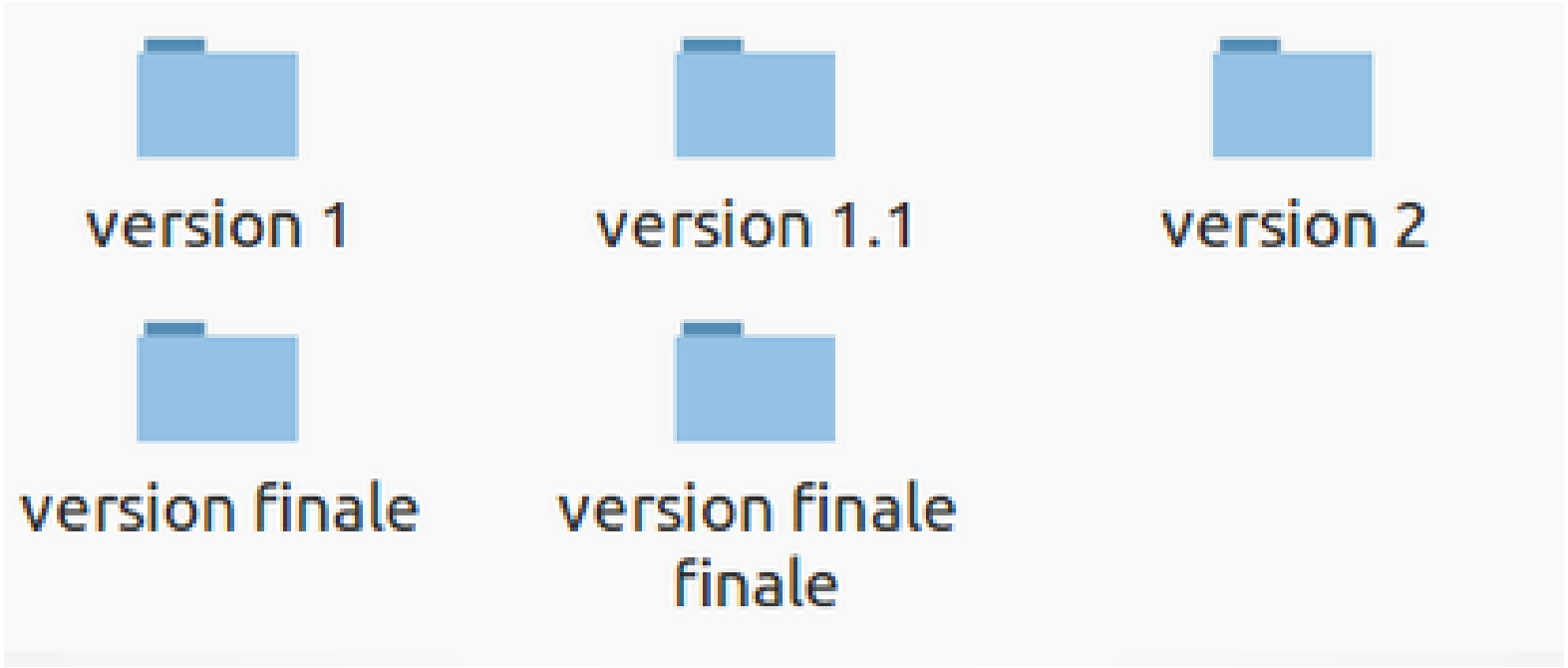


Figure: Avoiding the nightmare

Git

From its creator, Linus Torvalds[3]:

```
        GIT - the stupid content tracker

"git" can mean anything, depending on your mood.

 - random three-letter combination that is pronounceable, and not
   actually used by any common UNIX command.  The fact that it is a
   mispronounciation of "get" may or may not be relevant.
 - stupid. contemptible and despicable. simple. Take your pick from the
   dictionary of slang.
 - "global information tracker": you're in a good mood, and it actually
   works for you. Angels sing, and a light suddenly fills the room.
 - "goddamn idiotic truckload of sh*t": when it breaks

This is a stupid (but extremely fast) directory content manager.  It
doesn't do a whole lot, but what it _does_ do is track directory
contents efficiently.
```

Figure: GIT: *Global information tracker*

[3]Source: `https://github.com/git/git/blob/e83c5163316f89bfbde7d9ab23ca2e25604af290/README`

Initialization

We follow steps in the Github guide *Generating a new SSH key and adding it to the ssh-agent*[4]

- SSH Key
  1. Generate an SSH key (accept parameters by default, Don't introduce pass code)

```
$ ssh-keygen -t rsa -C "name.surname@ensta-paristech.fr"
```

  2. Show the generated key

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

  3. Paste the generated key in the Github interface, section 'My SSH Keys'. (one key required per computer you link to your github account)
- One time config

```
$ git config --global user.name "Diaz Natalia"
$ git config --global user.email "name.surname@ensta-paristech.fr"
```

[4] https://help.github.com/en/enterprise/2.16/user/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

A) Creating a project (when you have local work already)

1. Create a folder in your computer and initialize it

```
$ mkdir project_folder
$ cd project_folder
$ git init
```

2. Create a new project in GitHub.com GUI[5]
3. Add a new file

```
$ touch README.md
$ git add README.md
$ git commit -m "first commit"
```

4. Link your local folder to the Git project

```
$ git remote add origin git@github.com:ndiaz/project.git
```

5. Push (*upload*) the README.md over Git[6].

```
$ git push -u origin master
```

[5] https://help.github.com/en/articles/adding-an-existing-project-to-github-using-the-command-line

[6] `-u/ --set-upstream` adds an upstream (tracking) reference so to set origin as the upstream remote in your git config (this way you don't have to manually specify the remote every time you run git push, and so you can run git push without arguments)

A) Creating a project (when you have local work already)

- At this point, the project is created and initialized.
- Each person joining this project must be added as *collaborator* member through the Github interface, and simply should clone the project[7]:

```
$ git clone git@[srv_url]:[username]/[project].git
e.g.:
$ git clone git@github.com:ndiaz/project.git
```

[7] Prefer the SSH url address against the HTTPS one. The folder will be created in the location where you are located when launching this command

B) Creating a project (fastest)

- Create a new repo in Github.com GUI once logged in (Upper right '+' button)
- Add *collaborator* members through the Github interface, and simply clone the project[8]:

```
$ git clone git@[srv_url]:[username]/[project].git
e.g.:
$ git clone git@github.com:ndiaz/project.git
```

- Now you can create files inside the *project* folder and use the regular commands (from next slide)

[8] As in case A, Prefer the SSH url address against the HTTPS one. The folder will be created in the location where you are located when launching this command

Commands

- `Add`: adds file(s) for the next commit

```
$ git add my_file1 my_file2
$ git add --all
```

- `Commit`: *saves* files added previously

```
$ git commit -m 'Comment over the performed changes'
```

- `Pull`: get the changes others made

```
$ git pull
```

- `Push`: upload all changes on Git

```
$ git push
```

Example: common situation

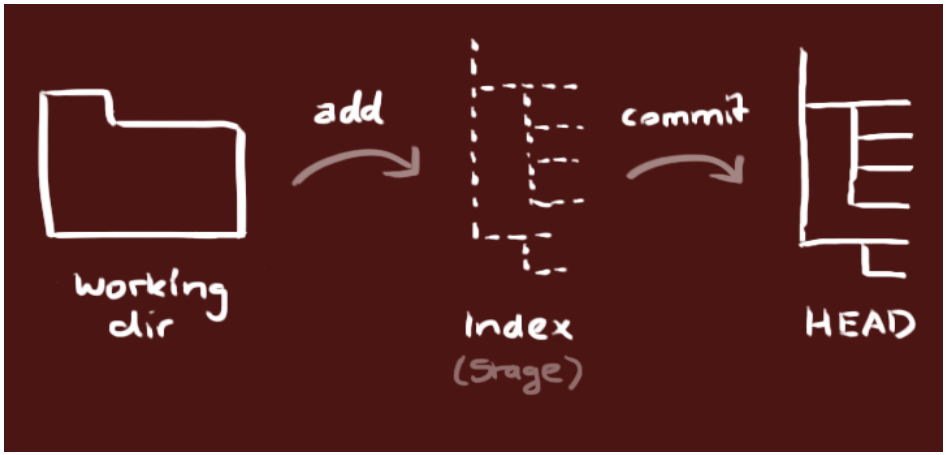
Example:
2 bugs to solve:
bug 1: requires modifying file a.py and b.py → bug 1 solved
bug 2: requires modifying file c.py → bug 2 solved

```
$ git add a.py b.py
$ git commit -m 'bug 1 solved!'
$ git add c.py
$ git commit -m 'bug 2 solved!'
$ git pull
$ git push
```

Commands

- `Status`: shows the status of the git local folder (modified/to add/staged files...)

```
$ git status
```



ALWAYS do *pull* before *push*!!

Branches

- **List** branches

```
$ git branch
* master
```

- `branch`: Creates a branch

```
$ git branch my_new_branch
$ git branch
* master
  my_new_branch
```

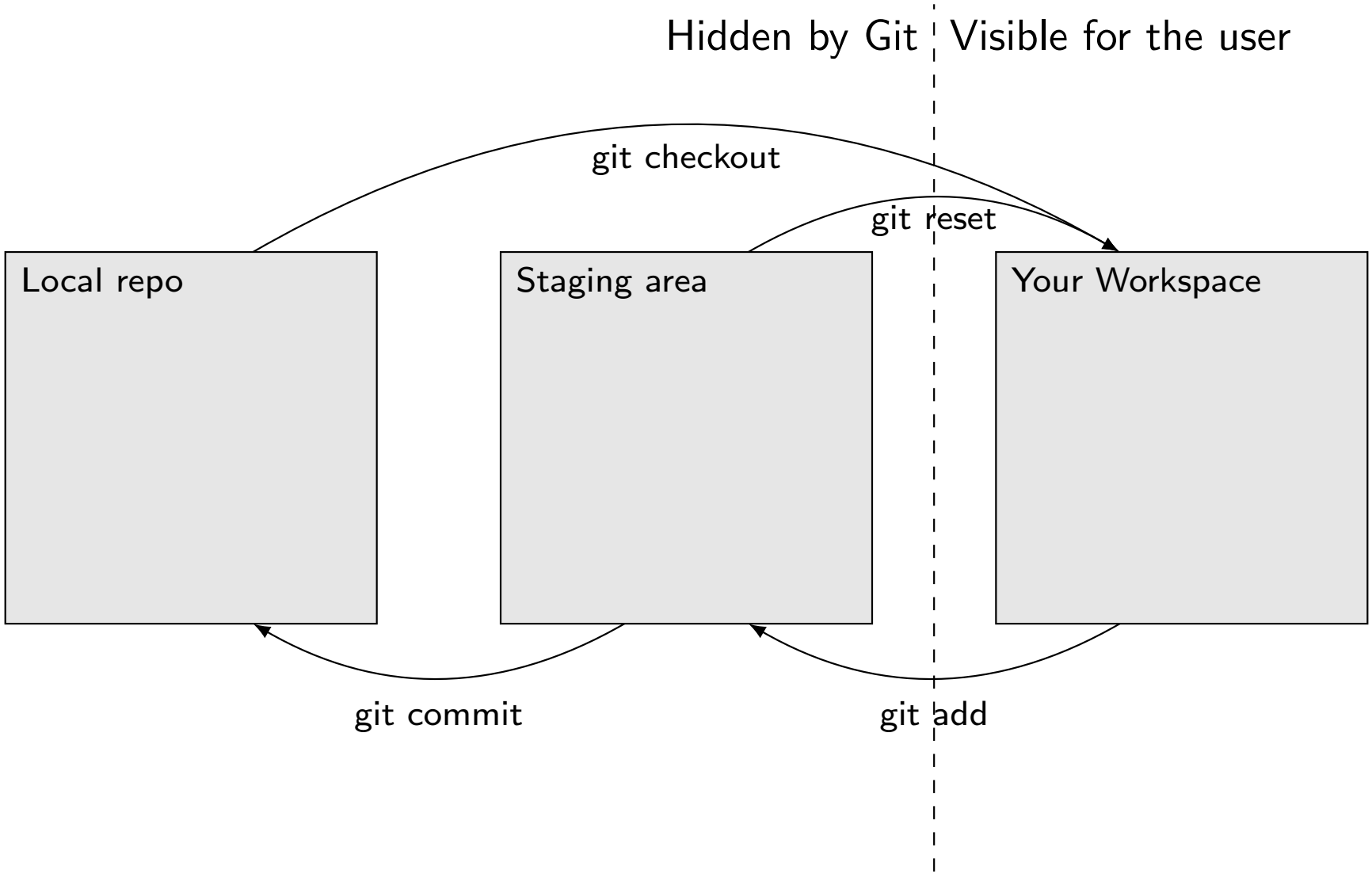
- `checkout`: Place yourself in my_new_branch

```
$ git checkout my_new_branch
```

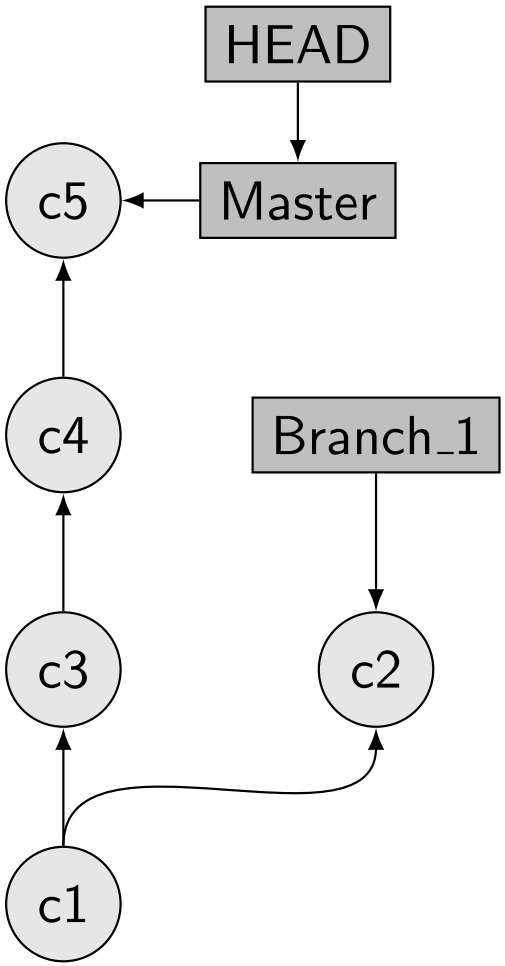
- `Merge`: Fuses my new branch into master

```
$ git checkout master
$ git merge my_new_branch
```

Git Workspace



Branches, visually: commits history

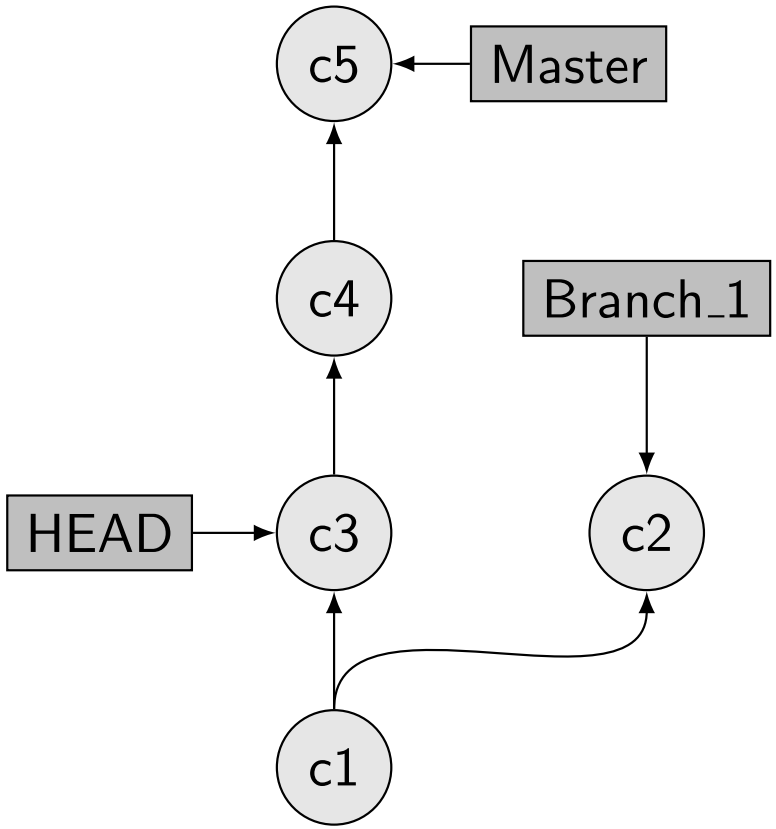


- Rectangle = branch.
- Circle = commit.
- **HEAD**: a ref. to the last commit in the currently checked-out branch (think of it as the *current branch*): When you switch branches with git checkout, the HEAD revision changes to point to the tip of the new branch[a].
- **master**: the default branch created when you init a git repo. You can delete the master branch but you can't delete the HEAD pointer.

[a] You can see what HEAD points to by doing:

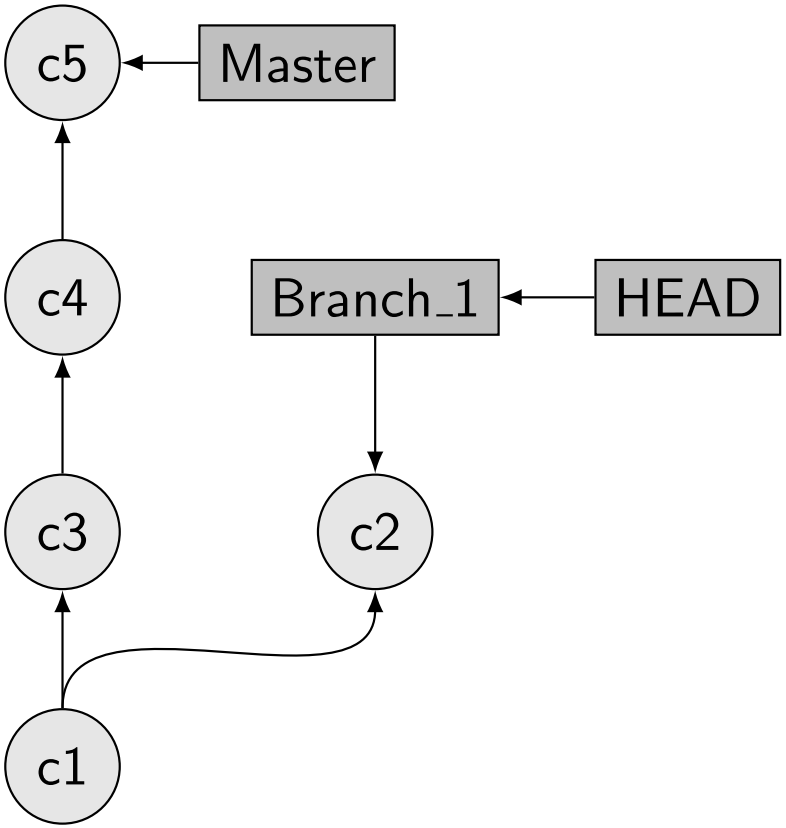
```
cat .git/HEAD
```

Branches, visually: commits history



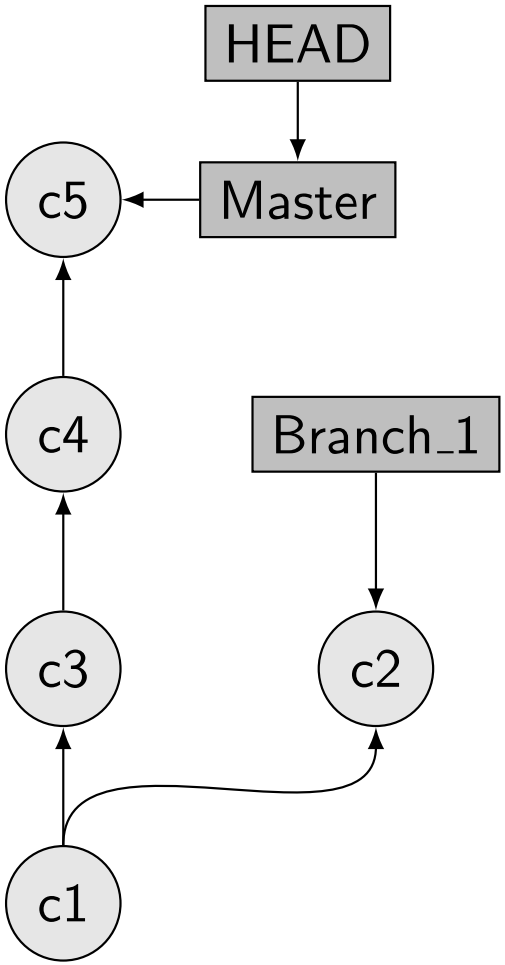
$ git checkout c3

Branches, visually: commits history



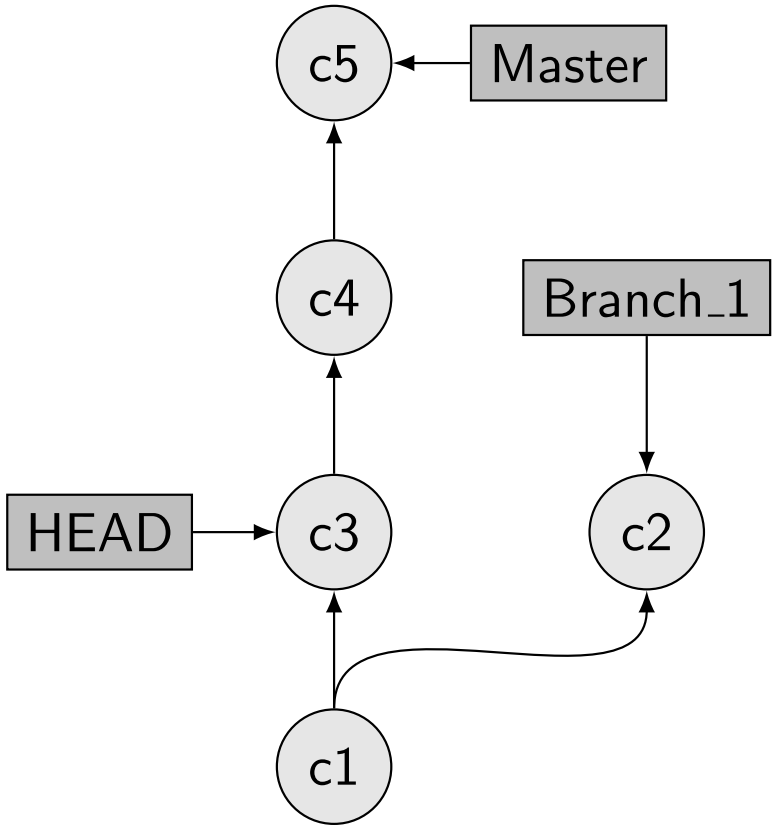

```
$ git checkout c3
$ git checkout Branch_1
```

Branches, visually: commits history




```
$ git checkout c3
$ git checkout Branch_1
$ git checkout master
```

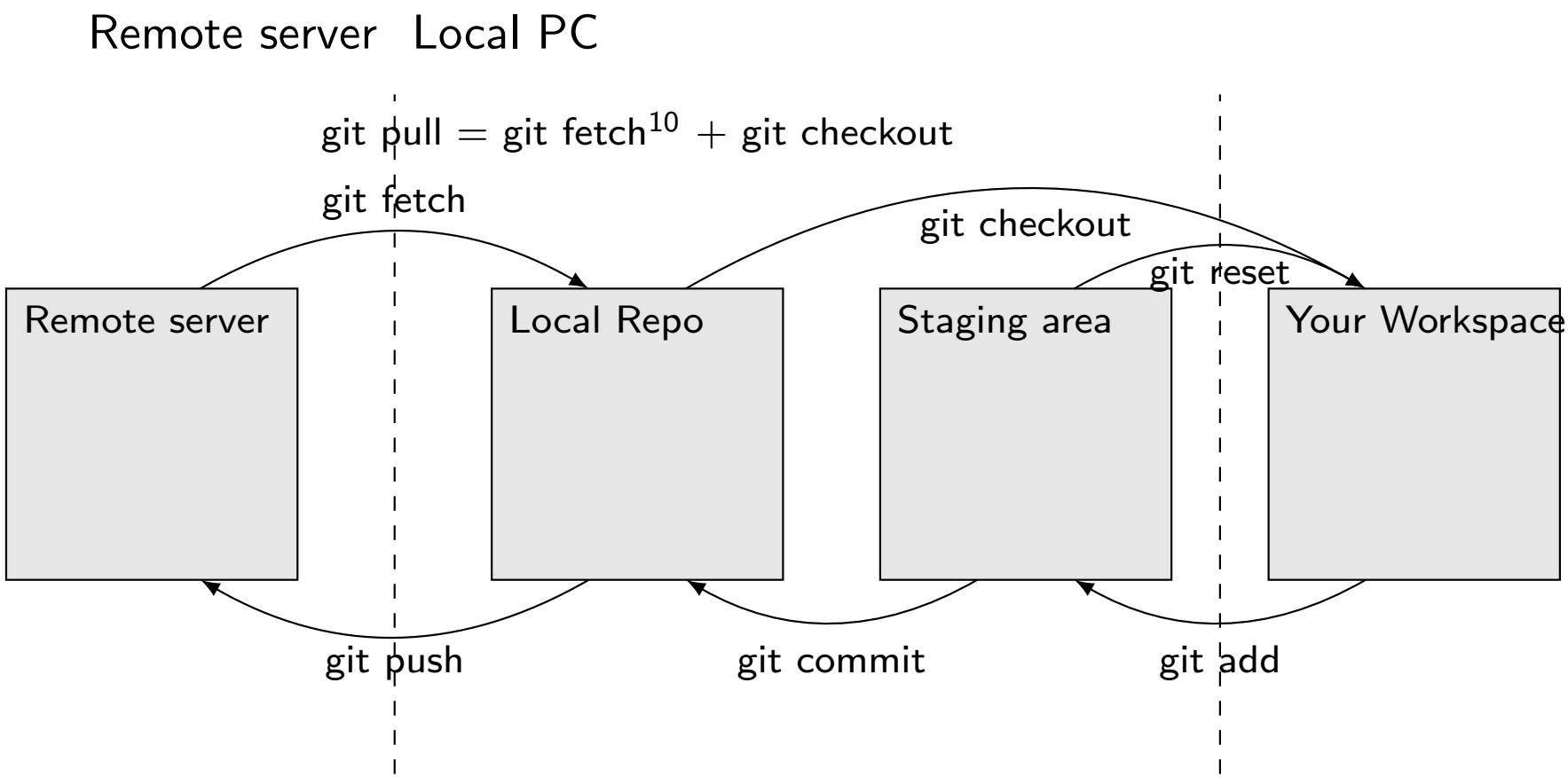
Branches, visually: commits history[9].




```
$ git checkout c3
$ git checkout Branch_1
$ git checkout master
$ git checkout master~2
```

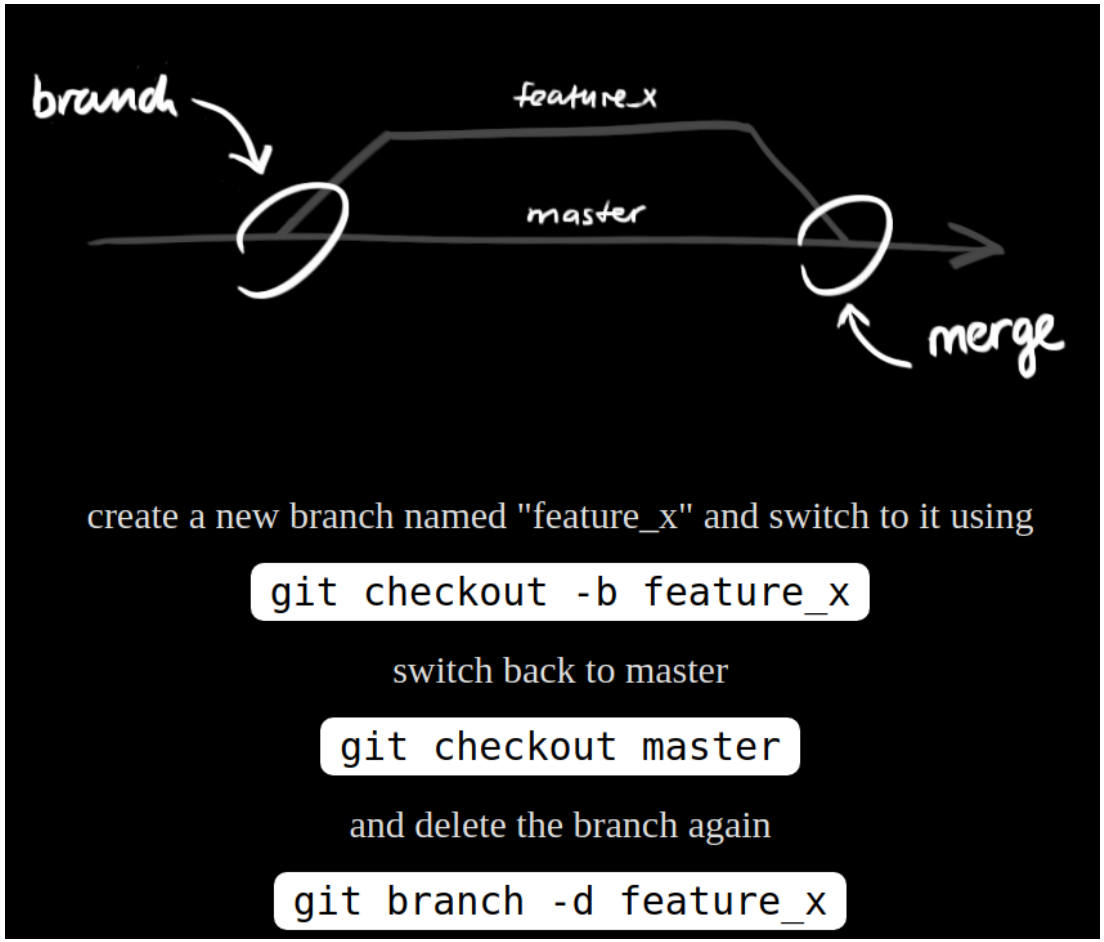
[9] `git checkout <tag>~n` means "checkout to the commit in the n-th position behind <tag>"

Link with a remote repository

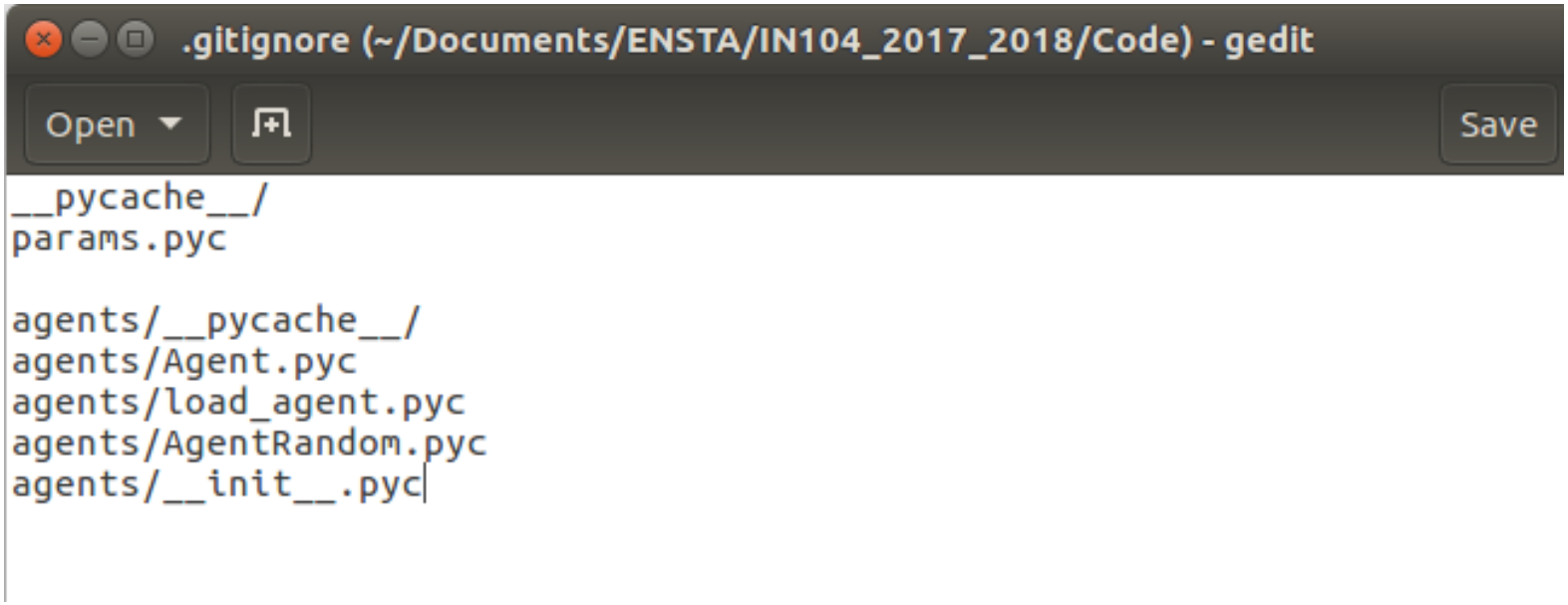


[10] git fetch vs git pull `https://www.atlassian.com/git/tutorials/syncing/git-fetch`

Creating and deleting branches **Re-cap**



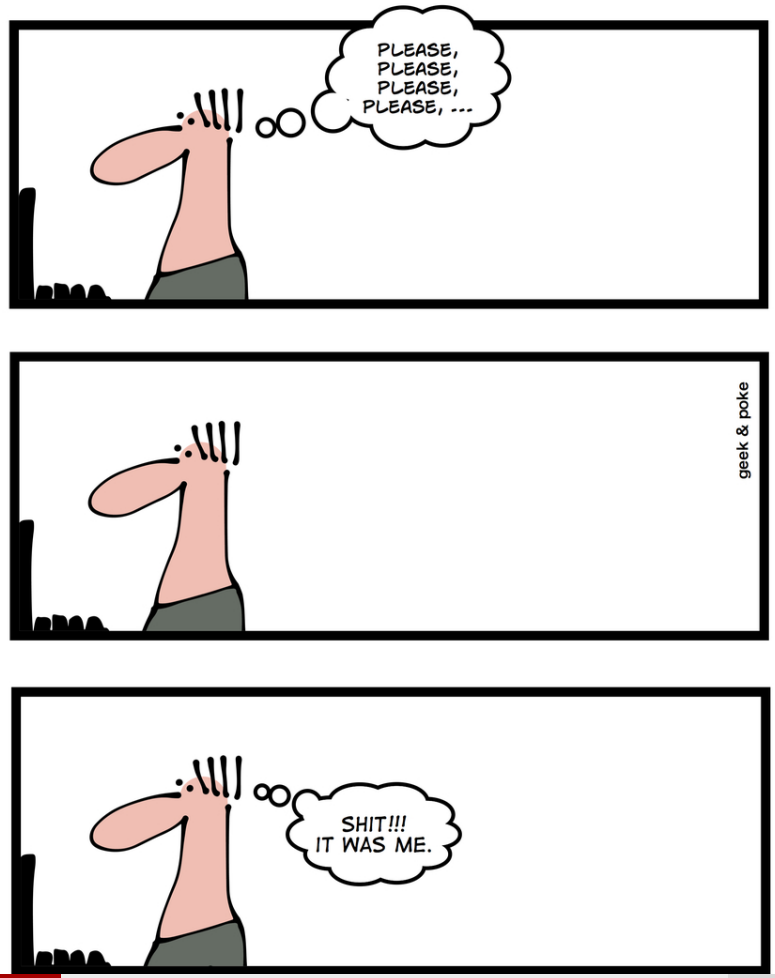
Ignore files we don't want uploaded in Git: **.gitignore**



- Avoids uploading unnecessary compilation/intermediate files to Git.
- List of .gitignore templates for a broad list of languages:
  `https://github.com/github/gitignore`
  `https://www.gitignore.io/`

Git blame

Who introduced this bug?



Git blame

Catkin_OpenCV.txt 2.18 KB

Edit | Raw | Blame | History | Permalink | Remove

Catkin_OpenCV.txt 2.18 KB

Edit | Raw | Normal View | History | Permalink | Remove

e85d9861 Carton Florence first tests with ...	1	Catkin avec OpenCV
	2	
	3	
	4	Creation du catkin workspace
	5	
	6	$ mkdir -p ~/catkin_ws/src
	7	(ou $ mkdir /catkin_ws/src)
	8	$ cd /catkin_ws/src
	9	$ catkin_init_workspace
	10	
20f78886 Carton Florence 1er tests avec op...	11	Creation d'un package catkin (tjs dans le dossier src de catkin_ws)
e85d9861 Carton Florence first tests with ...	12	
	13	$ catkin_create_pkg ardrone_vision std_msgs roscpp
	14	
	15	ardrone_vison : name of the package

Merge conflict

```
$ git merge my_branch
Auto-merging test_file.md
CONFLICT (content): Merge conflict in test_file.md
Automatic merge failed; fix conflicts and then commit the result.
```

Inside test_file.md you will see:

```
<<<<<<< HEAD
# some modifications here created conflicts
=======
# blablabla! breaking some code blabla
>>>>>>> my_branch
```

→ **HEAD**: modifications in master branch
→ **my_branch**: modifications of my_branch (others)
Edit it to keep the right changes. Once problems are solved:

```
$ git add test_file.md
$ git commit -m 'Solved merge conflict in test_file.md'
$ git push
```

Going back in time: recovering a past version

- Abandon changes done in a particular file

```
$ git checkout -- my_file
```

- Cancel the changes done in last commit

```
$ git revert
```

Going back in time: recovering a past version

Panic mode?

If you get stuck with a bunch of unintentional merge errors and want to reset your repo:

```
git fetch origin
git reset --hard origin/master
```

Note that you will lose EVERYTHING unsaved (or maybe even saved) in your repo! Keep a backup copy.

Practical time!

In the lab you will:

1. Learn GIT through the excellent GitHub Hello World Guide[11], GitHub Flow Guide[12] and GitHub Handbook Guide[13].
2. Create a PRIVATE repository called *IN104_NameA_SurnameA_NameB_SurnameB* (for all team members, max 3 members), add as collaborators your team mate(s) and your Teaching Assistant (TA). Create a folder inside called "GIT" that contains a sample hello_world.py Python program that your mate needs to modify, commit, and you need to retrieve the changes he did.
3. Show the program modified by both members and the commits in github by both team partners to your TA
4. Send the link to your repository to your TA (finish at home within 1 week if no time in class!)

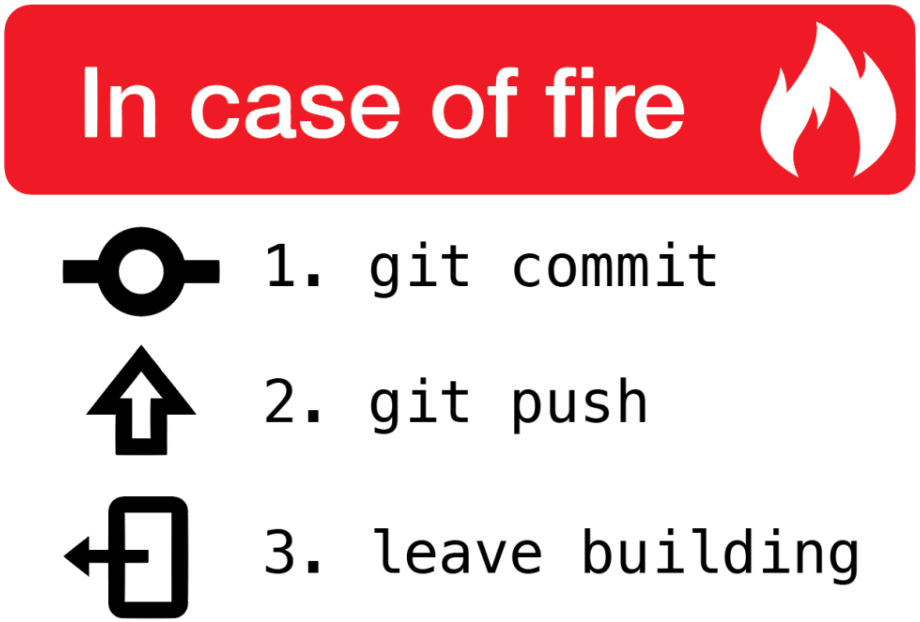
[11] `https://guides.github.com/activities/hello-world/`
[12] `https://guides.github.com/introduction/flow`
[13] `https://guides.github.com/introduction/git-handbook/`

Practical time!

1. The same game of GIT commits in your collaborative team project will be evaluated in your final repository
2. If you finish on time, play more advanced GIT in `https://gitexercises.fracz.com` and `https://www.codecademy.com/courses/learn-git/lessons/git-branching/exercises/branching-overview`
3. Q: Should I use Gitlab or Github?
   A: We strongly encourage the use of GitHub. If you really really want to use Gitlab, use `gitlab.ensta.fr` and set up your account and SSH Keys as in here[14]

[14] `https://gitlab.com/help/ssh/README`

To Conclude



Useful links

- First time user/computer: Generating a new SSH key and adding it to the ssh-agent[15]
- GIT Cheat Sheets:
  https://education.github.com/git-cheat-sheet-education.pdf
  https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet
  In French: https://github.com/UgoVollhardt/CheatSheetGit/blob/master/CheatSheet.pdf
- Oh shit git! http://ohshitgit.com/
- How to undo (almost) everything in Git https://blog.github.com/2015-06-08-how-to-undo-almost-anything-with-git/
- Openclassroom: Manage your source code with Git and Github (in FR):
  www.openclassrooms.com/courses/gerer-son-code-avec-git-et-github

[15] https://help.github.com/en/enterprise/2.16/user/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

Useful links

Interactive tutorials to learn by doing:

- https://gitexercises.fracz.com
- https://www.codecademy.com/courses/learn-git/lessons/git-branching/exercises/branching-overview
- https://learngitbranching.js.org/
- https://try.github.io/levels/1/challenges/1

Per-command Atlassian guide (e.g. checkout vs fetch vs pull):

- https://www.atlassian.com/git/tutorials/syncing/git-fetch

Useful links

- Antonin Raffin tutorials - Intro to Git: http://slides.com/antoninraffin/git and Git intermediate: http://slides.com/antoninraffin/git-intermediate
- http://users.humboldt.edu/smtuttle/s12cis492/492guide-to-git.pdf
- https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf
- https://github.com/git-tips/tips#everyday-git-in-twenty-commands-or-so
- https://tutorialzine.com/2017/11/10-useful-git-tips

Useful links: Going beyond

- Install Python libraries and Master Python: `http://musicinformationretrieval.com/python_basics.html`
- Python Numpy `http://cs231n.github.io/python-numpy-tutorial/` and IPython tutorials `http://cs231n.github.io/ipython-tutorial/`
- Iterate fast installing Jupyter notebooks `http://jupyter.org/install` and get good at IPython: `http://musicinformationretrieval.com/get_good_at_ipython.html`
- The quartet of NumPy, SciPy, Matplotlib, and IPython is a popular combination in the Python world. Numpy Basics: `http://musicinformationretrieval.com/numpy_basics.html`
- Numpy Tutorial: `http://scipy.github.io/old-wiki/pages/Tentative_NumPy_Tutorial`

Appendix


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

FAQ

- Q: git merge error

```
$ Merge branch 'master' of github.com:NataliaDiaz/repo-name
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

A: To solve it in linux: Ctrl+X (Exit). In mac:
press "i"
write your merge message
press "esc"
write ":wq"
then press enter
You should see something like:

```
Merge made by the 'recursive' strategy.
 YourChangedFile.txt | 57 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 1 file changed, 57 insertions(+)
```

FAQ

- Q: First time pull:

```
git pull
There is no tracking information for the current branch. Please specify
    git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so w

    git branch --set-upstream-to=origin/<branch> master
```

A:

```
$ git branch --set-upstream-to=origin/master  master

$ git pull --allow-unrelated-histories
```

FAQ

- Q: First time push when associating local repo to a remote:

```
$ git pull
fatal: refusing to merge unrelated histories
```

A:

```
$ git pull --allow-unrelated-histories

Merge made by the 'recursive' strategy.
```

FAQ

Fetch vs Pull?

`Fetch`:

- Similar to pull, except it won't do any merging.
- Downloads commits, files, and refs from a remote repository into your local repo.
- What you do when you want to see what everybody else has been working on.
- Doesn't force you to actually merge the changes into your repository. Git isolates fetched content from existing local content
- Has absolutely no effect on your local development work.

`Checkout`:

- If done on a local copy of a remote branch, it creates a local copy of the branch and merges it in the local branch by default.

FAQ

Fetch vs Pull?

Example: Fetch will pull down the *remoteBranch* and create a local copy of a remote branch which you shouldn't manipulate directly; instead create a proper local branch and work on that.

```
$ git checkout localBranch
$ git fetch origin remoteBranch
$ git branch
    master
        * localBranch
        remoteBranch
```

Summary:

- pull = fetch + merge
- pull = fetch + checkout