

# Introduction à l'informatique : lignes de commandes et git

---

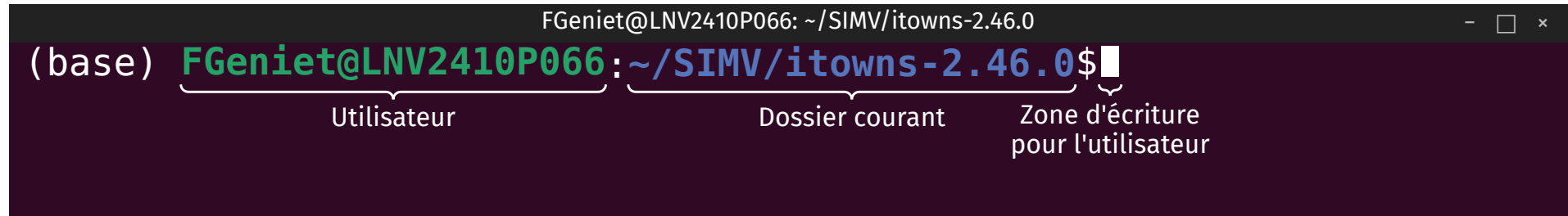
Florent GENIET

1re année géomatique  
22 septembre



# Terminal

Terminal : **interface** homme-machine **textuelle**



A screenshot of a terminal window with a dark background. The title bar at the top reads "FGeniet@LNV2410P066: ~/SIMV/itowns-2.46.0" and includes standard window control icons. The main content shows a shell prompt "(base) FGeniet@LNV2410P066: ~/SIMV/itowns-2.46.0\$". Below the prompt, three white curly braces with labels identify its components: "Utilisateur" under "FGeniet@LNV2410P066", "Dossier courant" under "~/SIMV/itowns-2.46.0", and "Zone d'écriture pour l'utilisateur" under the "\$" symbol. A white cursor is positioned at the end of the prompt.

```
FGeniet@LNV2410P066: ~/SIMV/itowns-2.46.0
(base) FGeniet@LNV2410P066: ~/SIMV/itowns-2.46.0$
```

Utilisateur                      Dossier courant                      Zone d'écriture  
pour l'utilisateur

# Le langage bash

Sur unix (linux et mac), le langage utilisé dans le terminal est **bash**.  
Les commandes bash s'appellent toujours de la même façon :

```
commande [-one_letter_options] [--other_options] <arguments>
```

Pour avoir la liste des options d'une commande :

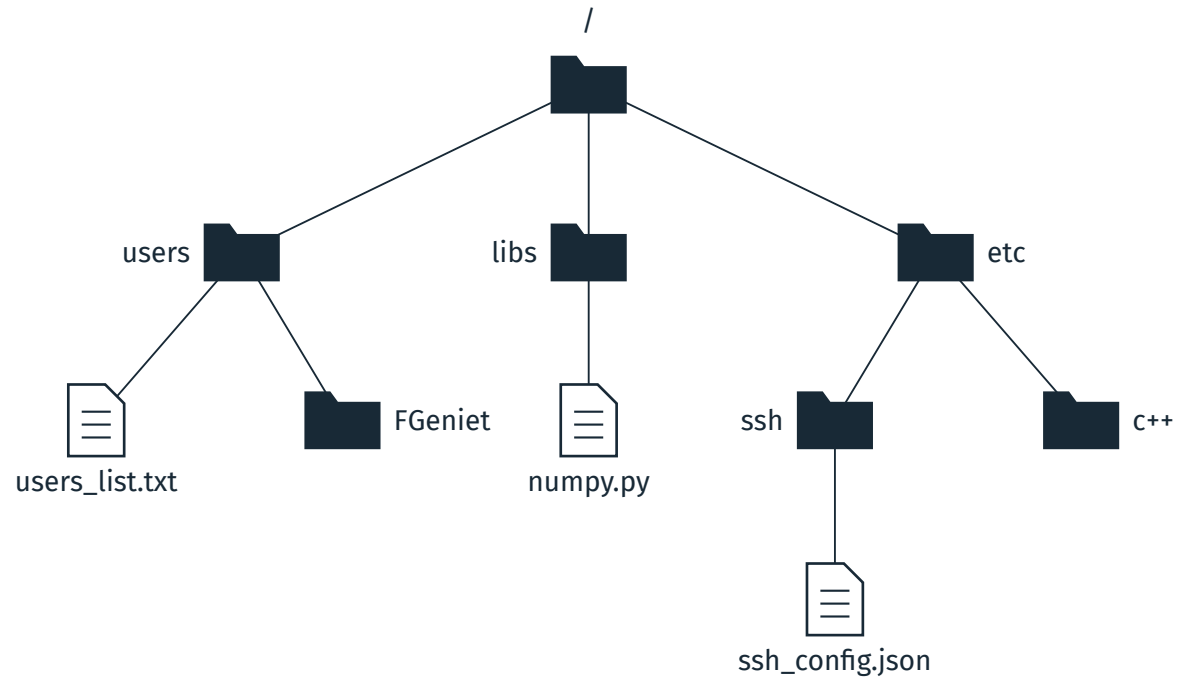
```
commande --help
```

# Liste de commandes utiles

```
#lister les fichiers et sous dossiers d'un dossier
ls [dossier]
#changer de dossier courant
cd <dossier>
#copier un fichier ou un dossier
cp [-r] <source> <destination>
#déplacer un fichier ou un dossier
mv <source> <destination>
#supprimer un fichier ou un dossier
rm [-r] <source>
#savoir dans quel dossier on se trouve
pwd
#créer un fichier
touch <nom_fichier>
#créer un dossier
mkdir <nom_dossier>
```

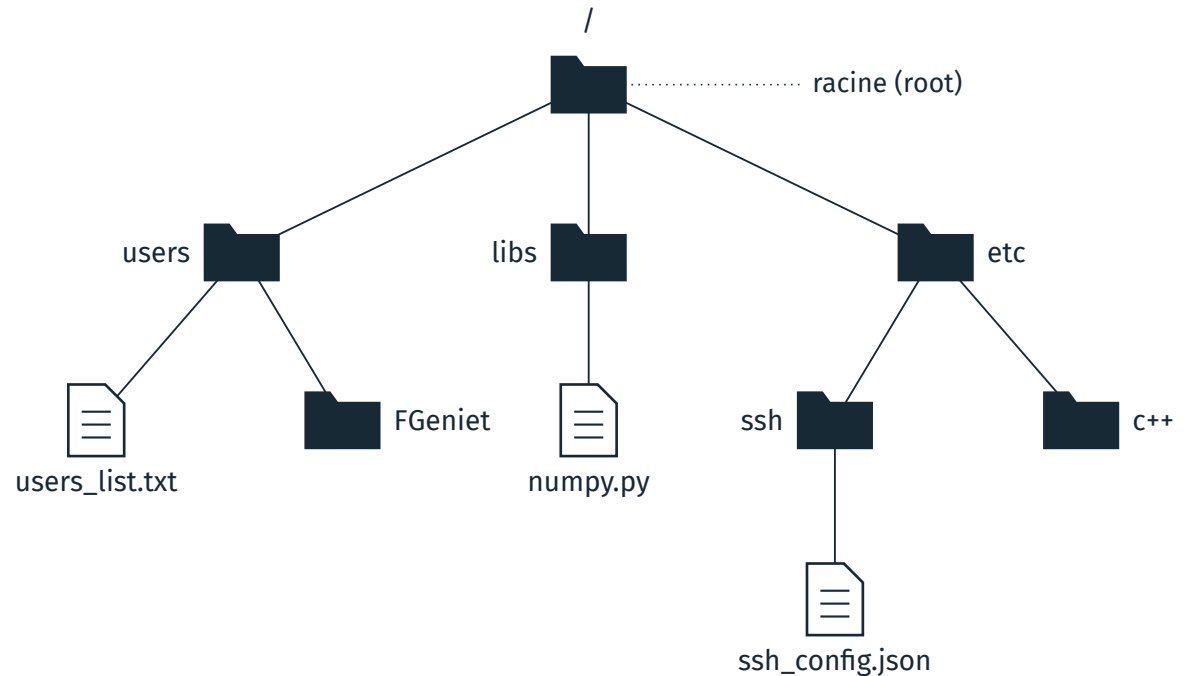
# Arborescence de fichiers

Les fichiers et dossiers sont organisés en **arbre**



# Arborescence de fichiers

**racine de l'arbre** : un dossier contenant tout.  
Chemin de la racine jusqu'à un dossier ou fichier = **chemin absolu**



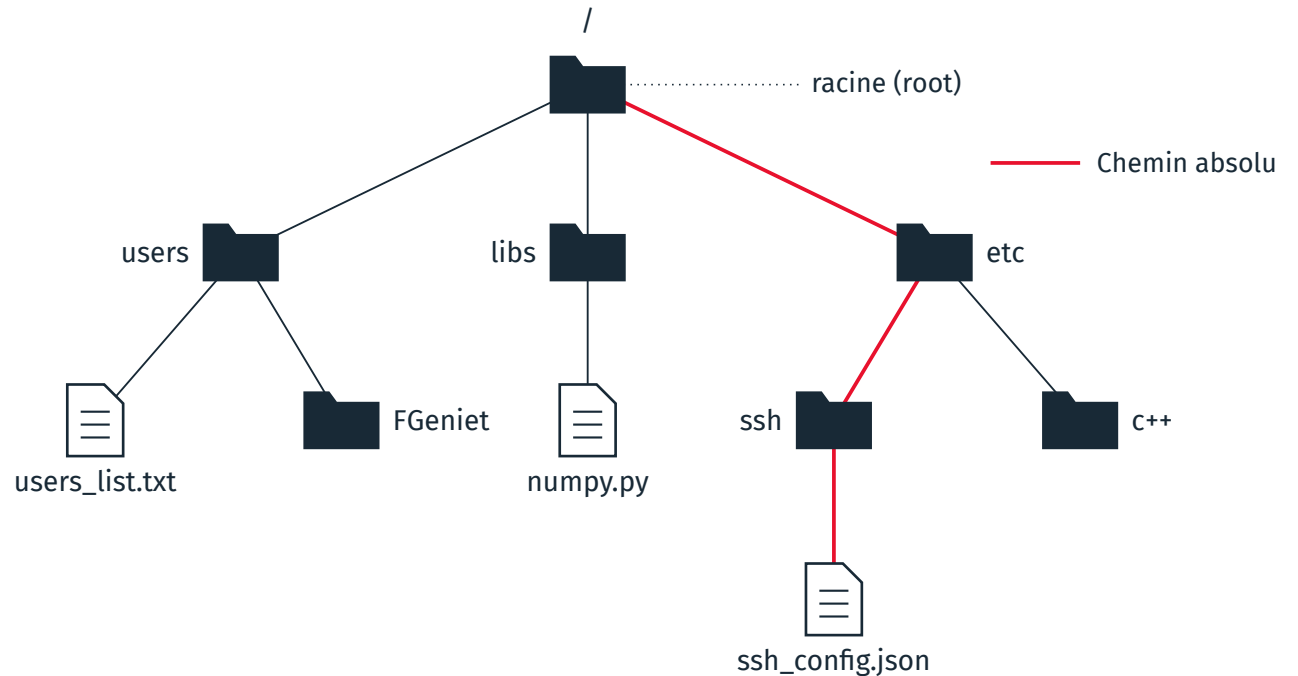
# Arborescence de fichiers

Le chemin absolu se note :

```
/dossier_1/.../  
dossier_n/destination
```

exemple :

```
/etc/ssh/  
ssh_config.json
```



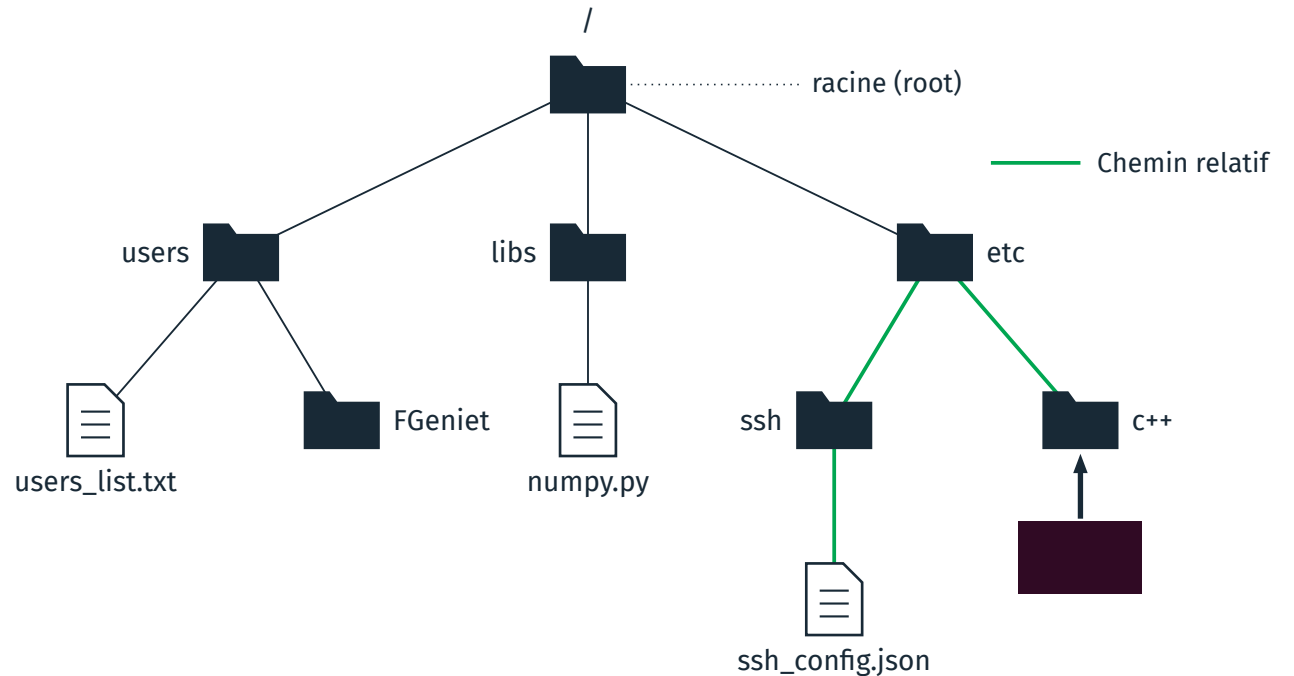
# Arborescence de fichiers

Chemin d'un dossier  
quelconque à un autre  
dossier ou fichier = **chemin  
relatif**

**Le dossier de départ est le  
dossier courant du  
terminal !**

Le chemin relatif se note :

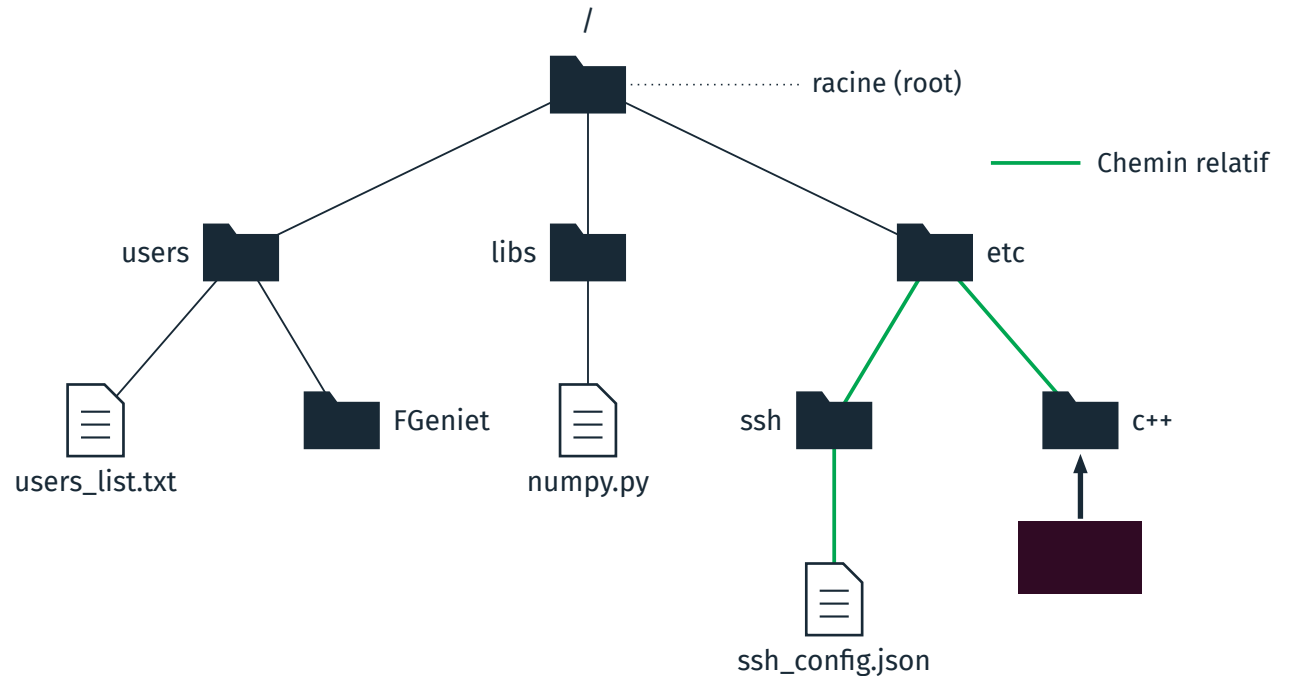
```
dossier_1/.../dossier_n/  
destination
```



# Arborescence de fichiers

Le dossier parent d'un autre dossier se note ".."  
exemple :

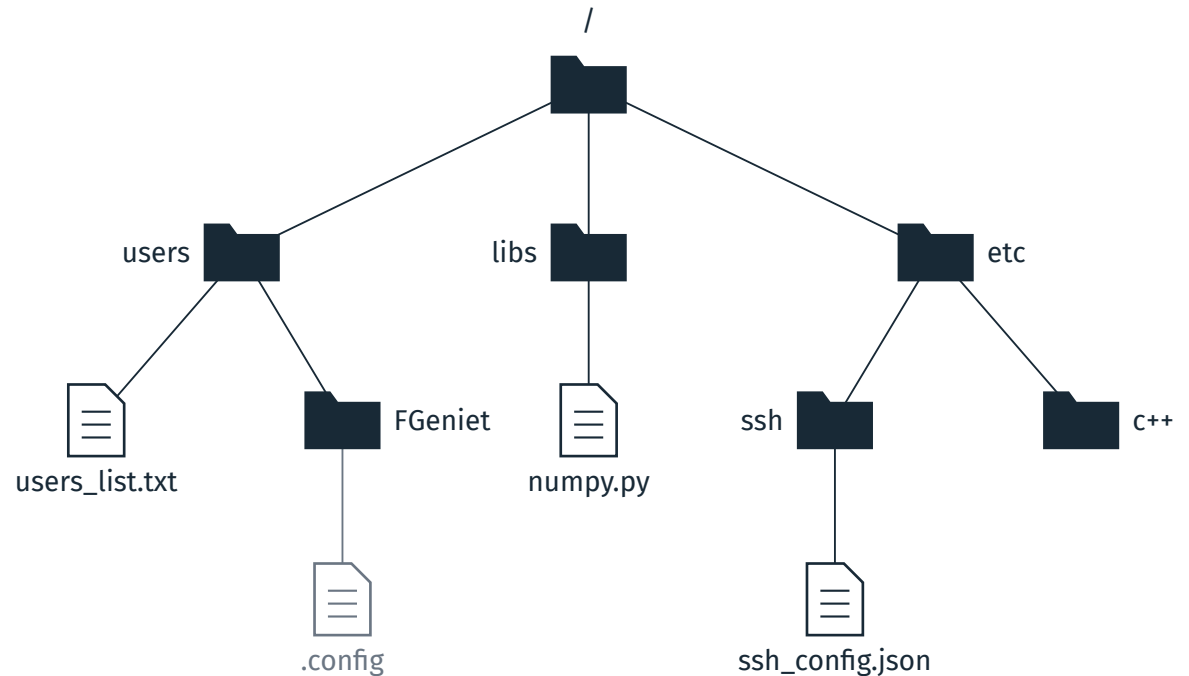
```
../ssh/ssh_config.json
```



# fichiers cachés

Certains fichiers/dossiers peuvent être **cachés**. Pour cacher un fichier/dossier, il faut que son nom commence par un ".".

Exemple : .git, .cache, .ssh



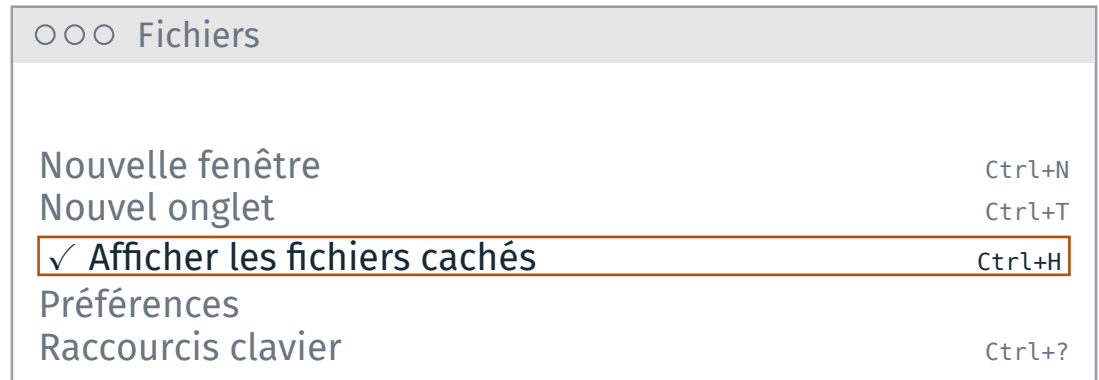
# fichiers cachés

Pour afficher les éléments cachés d'un dossier :

- En ligne de commande

```
ls -a [dossier]
```

- dans l'explorateur de fichiers :



# Expressions régulières

On peut désigner des groupes de fichiers/dossiers en utilisant des **expressions régulières (ou regex)**.

- "\*" : n'importe quelle chaîne de caractères
- exemples :
  - `/etc/*` : tous les dossiers et fichiers contenus dans etc
  - `/users/FGeniet/*.jpg` : toutes les images jpg du dossier FGeniet (ne comprend pas les images dans les sous-dossiers)
  - `*/*.pdf` : tous les pdf qui sont dans un sous-dossier du dossier courant

# Expressions régulières

On peut désigner des groupes de fichiers/dossiers en utilisant des **expressions régulières (ou regex)**.

- `"*"` : n'importe quelle chaîne de caractères
- `"?"` : n'importe quel caractère (un seul caractère)
- exemples :
  - `*_?.png` : tous les png avec un nom ayant un "\_" en avant dernier caractère
  - `/users/*/????.*` : tous les fichiers dans les dossiers utilisateurs ayant un nom de 4 caractères

# Expressions régulières

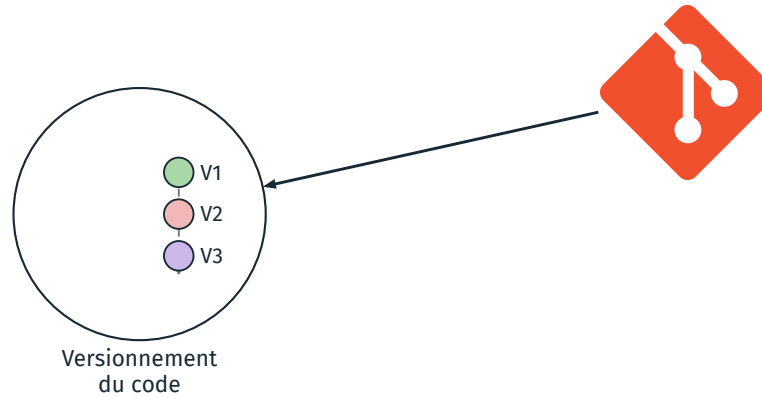
On peut désigner des groupes de fichiers/dossiers en utilisant des **expressions régulières (ou regex)**.

- `"*"` : n'importe quelle chaîne de caractères
- `"?"` : n'importe quel caractère (un seul caractère)
- `"["` : permet de donner une liste de caractères.
- exemples :
  - `mo[tn]o.jpg` : désigne `moto.jpg` et `mono.jpg`
  - `reunion_[0-9][0-9].txt` : les fichiers texte dont le nom est de la forme `reunion_` suivi d'un nombre à 2 chiffres

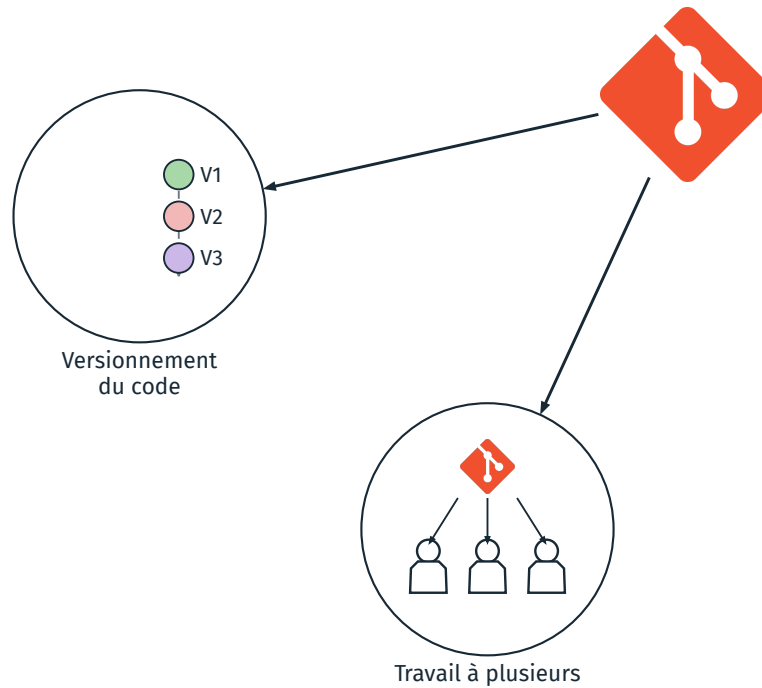
# Git : c'est quoi ?



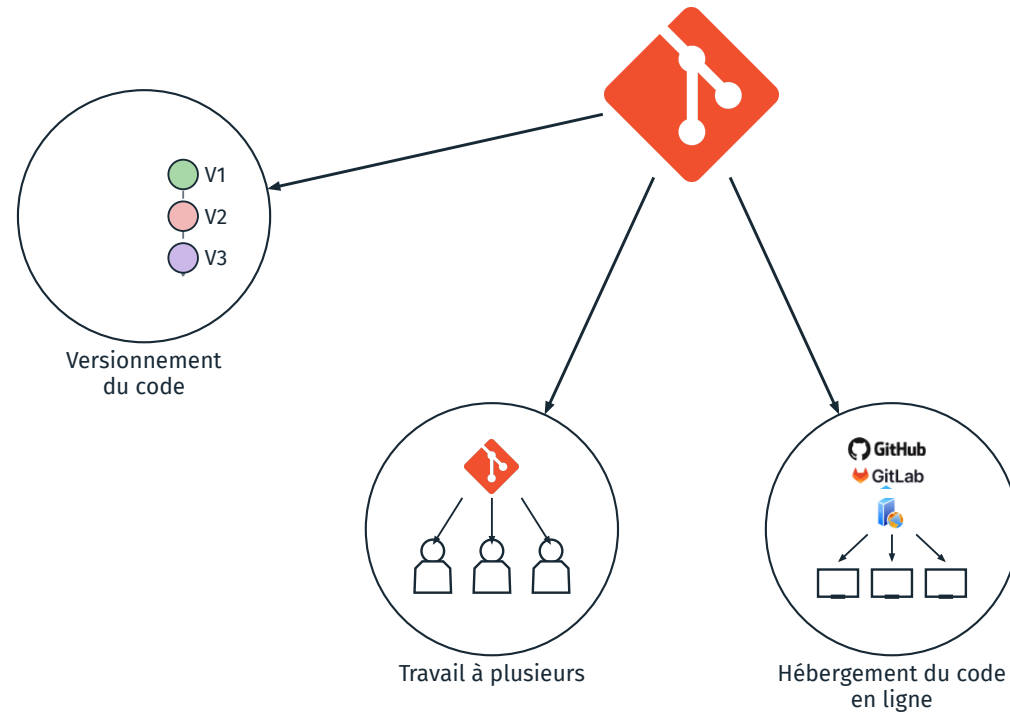
# Git : c'est quoi ?



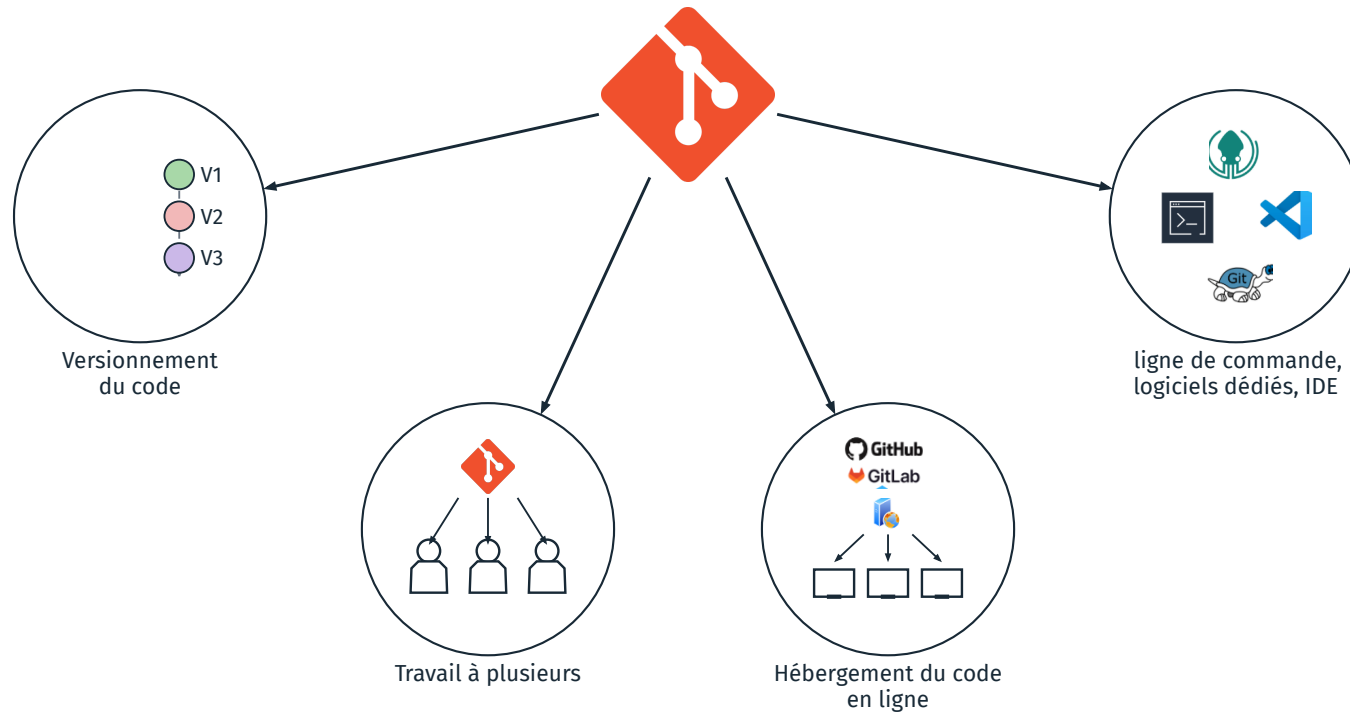
# Git : c'est quoi ?



# Git : c'est quoi ?



# Git : c'est quoi ?



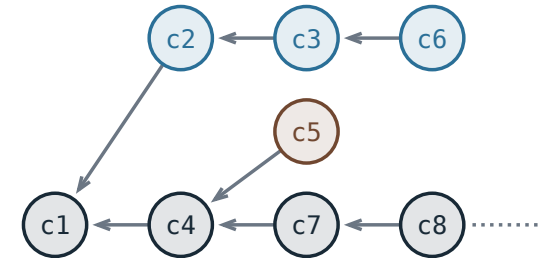
# Fonctionnalités

- Stockage des états du projet à chaque changement **enregistré**



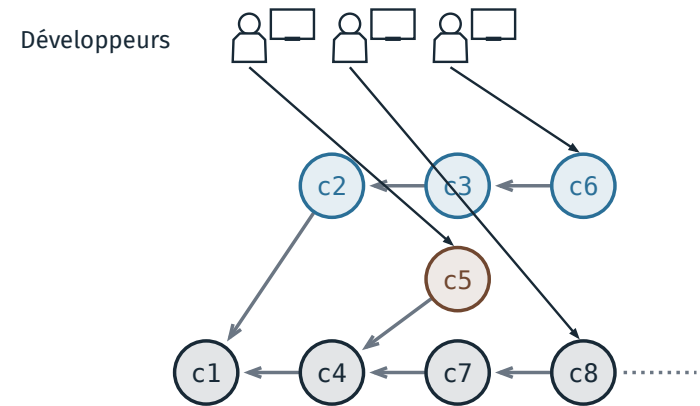
# Fonctionnalités

- Stockage des états du projet à chaque changement enregistré
- Modifications en **parallèle**



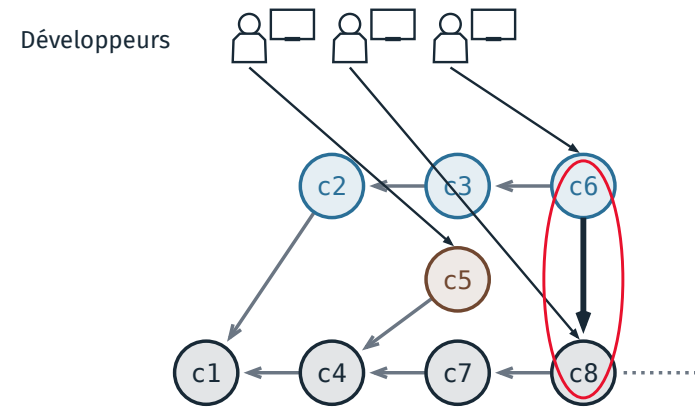
# Fonctionnalités

- Stockage des états du projet à chaque changement **enregistré**
- Modifications en **parallèle**
- Développement sur des **versions différentes**



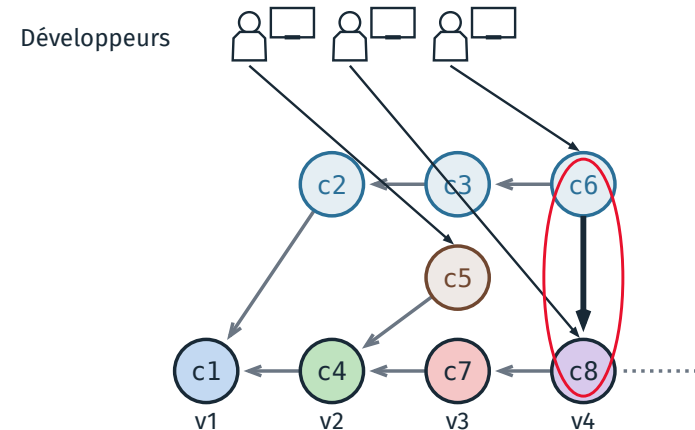
# Fonctionnalités

- Stockage des états du projet à chaque changement **enregistré**
- Modifications en **parallèle**
- Développement sur des **versions différentes**
- et **fusion** d'états modifiés parallèlement



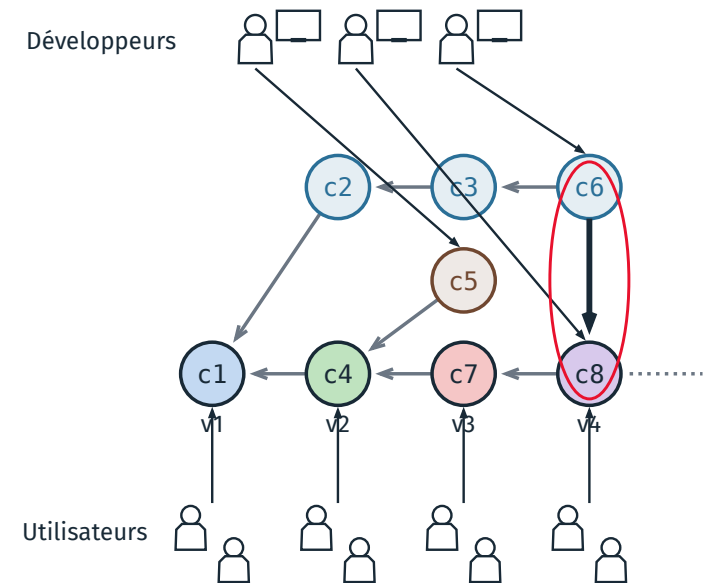
# Fonctionnalités

- Stockage des états du projet à chaque changement **enregistré**
- Modifications en **parallèle**
- Développement sur des **versions différentes**
- et **fusion** d'états modifiés parallèlement
- **Labellisation** de certains états (versions)



# Fonctionnalités

- Stockage des états du projet à chaque changement **enregistré**
- Modifications en **parallèle**
- Développement sur des **versions différentes**
- et **fusion** d'états modifiés parallèlement
- **Labellisation** de certains états (versions)
- Les utilisateurs ont accès aux différentes versions



# Git : quand l'utiliser ?

On utilise git pour gérer des **projets informatiques**.

- En projet solo :
  - Utilisation comme système de sauvegarde
  - Potentielle mise à disposition des versions finies
  - Structure git simple

# Git : quand l'utiliser ?

On utilise git pour gérer des **projets informatiques**.

- En projet solo :
- En projet à plusieurs :
  - Mise en commun du travail simplifiée
  - Travail sur des états parallèles  $\Rightarrow$  on ne se gêne pas
  - Application des bonnes pratiques en termes de structure git

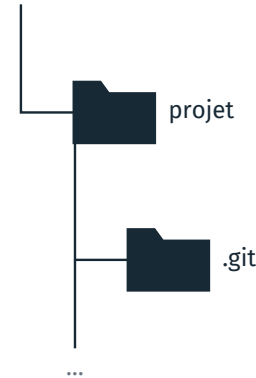
# Initialiser un projet git

Un projet git s'appelle un **repository** (ou **dépôt git** en français).

Créer un repository = créer un dossier *.git*

On utilise la commande suivante :

```
git init
```



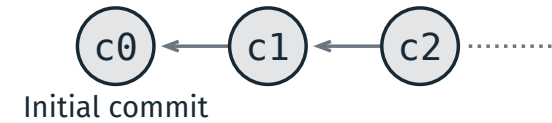
# Commit

Juste après `git init`, le dépôt ne contient aucun commit. Le premier commit s'appelle souvent **Initial commit**



# Commit

Chaque nouvel état enregistré du projet  
s'appelle un **commit**



# Enregistrer des modifications

Chaque fichier créé est **non suivi**.

Non suivi



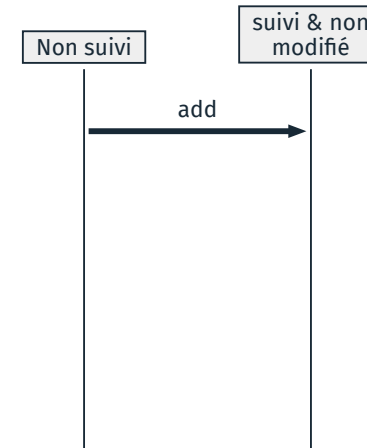
# Enregistrer des modifications

Pour commencer à suivre un fichier, on utilise la commande :

```
#ajouter un fichier  
spécifique  
git add <nom_de_fichier>
```

```
#ajouter un dossier et  
#tout ses fichiers  
git add <nom_de_dossier>
```

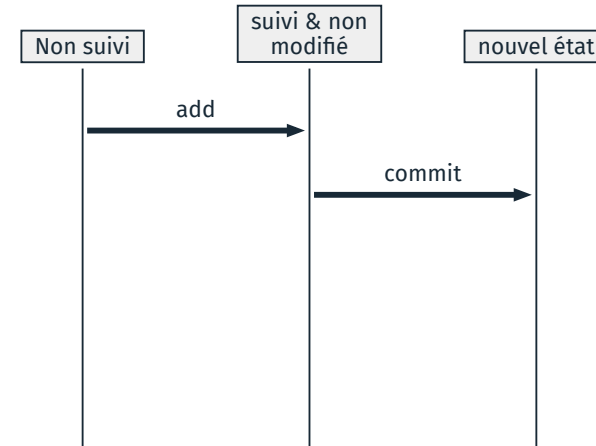
```
#ajouter le dossier courant  
git add .
```



# Enregistrer des modifications

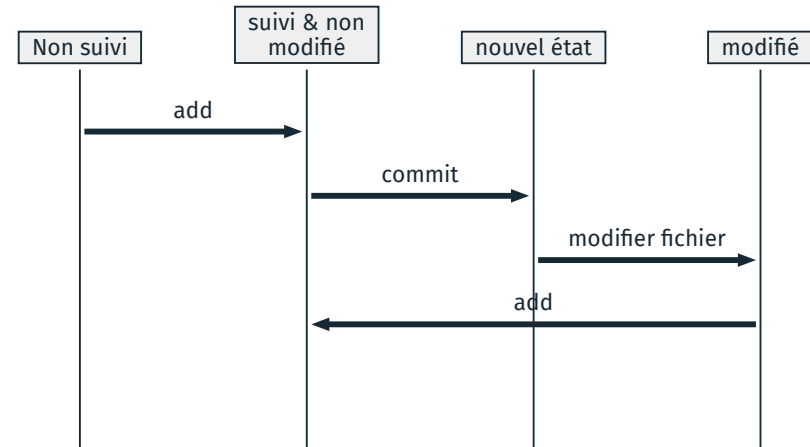
On peut ensuite créer un **nouvel état** du projet avec les nouveaux fichiers.

```
git commit -m  
<message_de_commit>
```



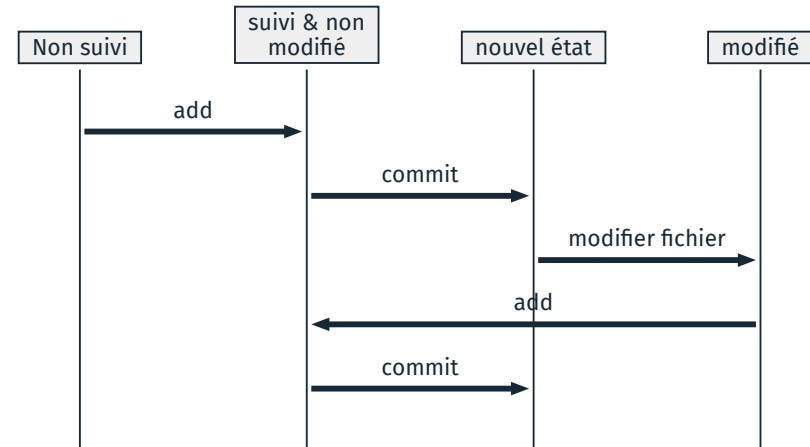
# Enregistrer des modifications

Pour valider un changement, on réutilise la commande **add**.



# Enregistrer des modifications

Il faut régulièrement faire des **commits**, car on ne peut naviguer qu'entre deux états enregistrés.

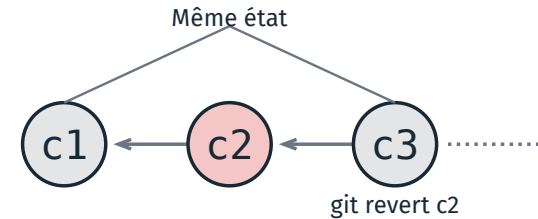


# Annuler un commit

On peut annuler un commit avec la commande :

```
git revert <nom du commit>
```

Cette commande crée un nouveau commit !



# Tagger un commit

On peut tagger un commit avec la commande :

```
git tag -a <nom du tag> -m <message de tag>
```



## TRAVAUX DIRIGÉS

---

**TD 2a**    Un premier dépôt  
cours2/2a\_premier\_depot/

≈ 15'

# Branches

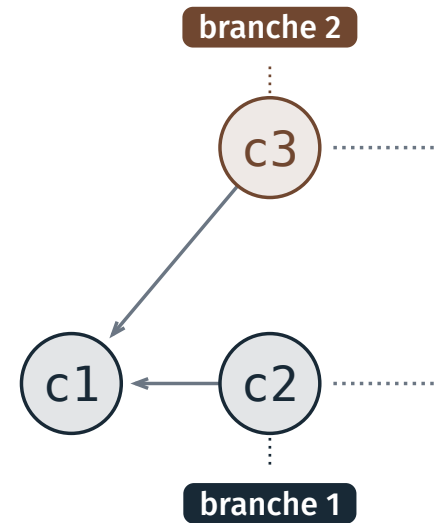
Création de **branches** de commits évoluant en parallèle.

```
#Créer une nouvelle branche  
#à partir du commit courant  
git branch <nom_de_branche>
```

```
#Créer une nouvelle branche  
#et s'y placer  
git checkout -b <nom_branche>
```

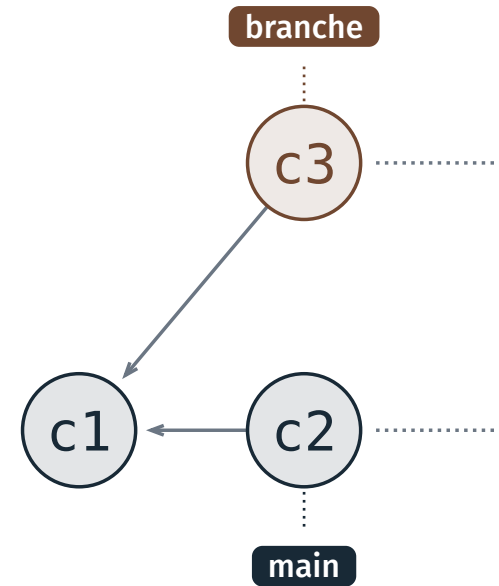
```
#Changer de branche  
git checkout <nom_de_branche>
```

```
#voir les différentes branches  
git branch
```



# Branches

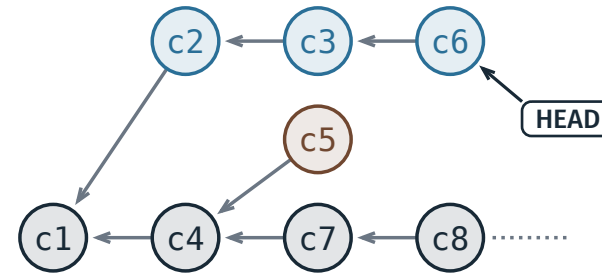
La branche initiale s'appelle **master**,  
ou parfois **main**.



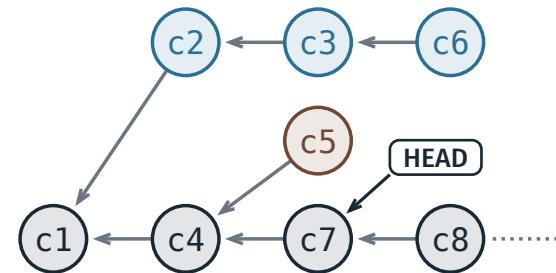
# Head

HEAD : désigne le commit sur lequel on se trouve (commit courant)

```
#Changer de commit  
git checkout  
<nom_de_commit>
```

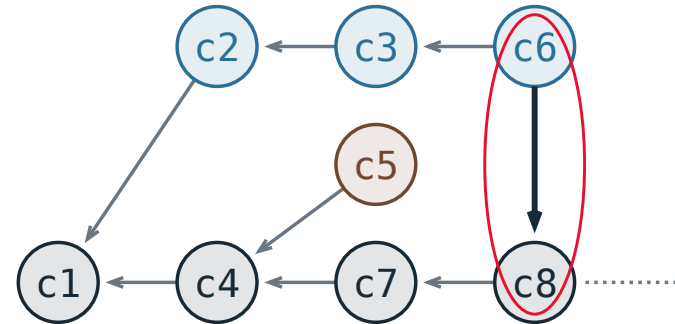


↓ git checkout commit\_7



# Fusionner deux branches

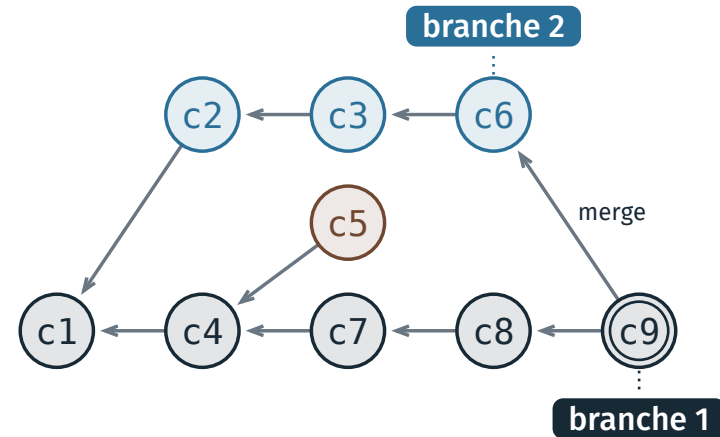
Deux branches peuvent être fusionnées  $\Rightarrow$  les modifications des deux branches sont ajoutées.



# Fusionner deux branches

Une première façon de fusionner est le **merge**.  
Le merge crée un nouveau commit de fusion

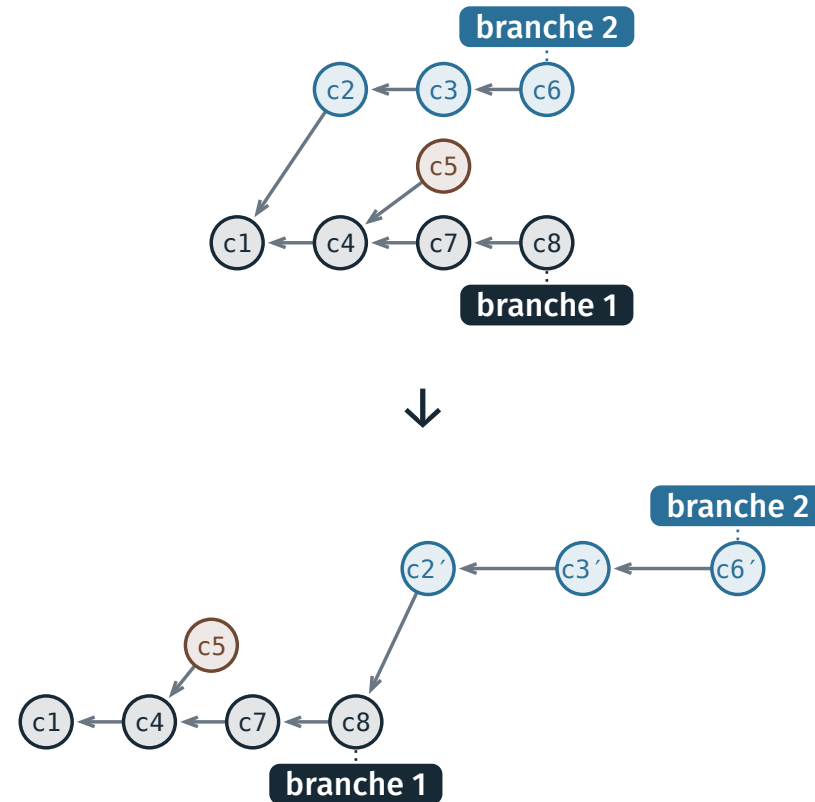
```
#Merge la branche b_1  
#dans la branche courante  
git merge <b_1>
```



# Fusionner deux branches

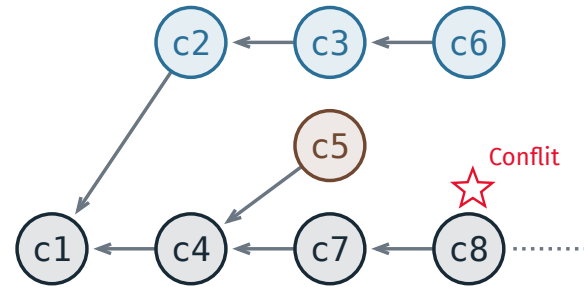
Une seconde façon de fusionner est le **rebase**.  
Le rebase rajoute les commits de la branche source au sommet de la branche cible.

```
#Refait les commits de  
b_2  
#au bout de b_1  
git rebase <b_1> <b_2>
```



# Conflicts

Deux modifications peuvent être contradictoires. Il y a alors **conflit** : le développeur doit choisir comment fusionner les deux modifications.



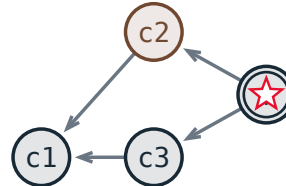
```
Fusion automatique de file.py
CONFLIT (contenu) : Conflit de fusion dans file.py
La fusion automatique a échoué ; réglez les conflits et validez
le résultat.
```

# Conflits

En cas de conflit lors d'un **merge**, git suspend le merge et marque les conflits **directement dans les fichiers.**

file.py

```
def fonction_1(a,b):  
    return a*b
```



file.py

```
def fonction_1(a,b):  
    return a+b
```

file.py

```
def fonction_1(a,b):  
    <<<<<< HEAD  
        return a+b  
    =====  
        return a*b  
    >>>>>> branche_1
```

# Conflicts

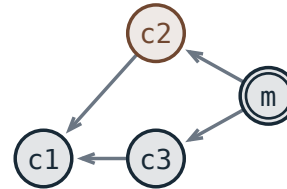
Les conflits se résolvent directement dans les fichiers. Une fois les conflits résolus, `git add` marque chaque fichier, puis on reprend le merge avec la commande

```
git add <fichier>
git merge --continue

#Idem pour le rebase
git rebase --continue
```

file.py

```
def fonction_1(a,b):
    return a*b
```



file.py

```
def fonction_1(a,b):
    return a+b
```

file.py

```
def fonction_1(a,b):
    return [a+b,a*b]
```

## TRAVAUX DIRIGÉS

---

|              |                                                                     |       |
|--------------|---------------------------------------------------------------------|-------|
| <b>TD 2b</b> | Branches et fusions<br><a href="#">cours2/2b_branches/</a>          | ≈ 30' |
| <b>TD 2c</b> | Annuler et remettre à jour<br><a href="#">cours2/2c_annuler/</a>    | ≈ 15' |
| <b>TD 2d</b> | Créer et résoudre un conflit<br><a href="#">cours2/2d_conflits/</a> | ≈ 25' |

# Afficher les informations

Le graphe git peut être affiché avec la commande :

```
git log --graph --  
pretty=oneline --abbrev-  
commit --decorate
```

```
* 468a7db Merge branch 'main' of https://github.com/.../2D-Viewer  
|\n| * e33576f Update package.json  
| * 7aabb31 fixing label rendering on prod  
| * 97felb2 Merge branch 'hotfix' into dev  
| |\n| | * dd9c38a (origin/hotfix, hotfix) fixing label rendering  
| | * | 932a982 package.json fix  
| | * | e6c32bc version with exact numbers  
* | | 56840cf Adding exact-math library  
| |/  
| * c0f07a8 Merge branch 'exact_math' into dev  
| * a4d7e7b (origin/exact_math) fix on the face choices  
| * c48800f input data correction algorithm  
|/  
* 1705866 Correction of the input data
```

# Afficher les informations

Pour voir la différence entre deux commits :

```
git diff <commit_1> [commit_2]
```

```
diff --git a/file.py b/file.py
index 6072655..e69de29 100644
--- a/file.py
+++ b/file.py
@@ -1,2 +0,0 @@
-def fonction_1(a,b):
-    return a+b
```

# Afficher les informations

Pour voir l'état du projet (ce qui est enregistré/pas enregistré) :

```
git status
```

```
Sur la branche master
```

```
Modifications qui ne seront pas validées :
```

```
(utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
```

```
(utilisez "git restore <fichier>..." pour annuler les modifications  
dans le répertoire de travail)
```

```
    modifié :      file.py
```

```
aucune modification n'a été ajoutée à la validation (utilisez "git add"  
ou "git commit -a")
```

# ignorer des fichiers

Pour demander à git d'ignorer des fichiers, il est possible de créer un fichier **.gitignore**.

On peut ensuite lister les fichiers à ignorer dans ce fichier

```
cache  
build  
img/*.png
```

# Bonnes pratiques

On va essayer de suivre certaines règles lorsqu'on utilise git, pour toujours avoir un git lisible et compréhensible.

- toujours décrire ce qu'on a fait dans le message de commit :

```
git commit -m "ajout de fonctionnalités dans le fichier..."
```

# Bonnes pratiques

On va essayer de suivre certaines règles lorsqu'on utilise git, pour toujours avoir un git lisible et compréhensible.

- toujours décrire ce qu'on a fait dans le message de commit :
- ne pas commit un code qui n'est pas fonctionnel

# Bonnes pratiques

On va essayer de suivre certaines règles lorsqu'on utilise git, pour toujours avoir un git lisible et compréhensible.

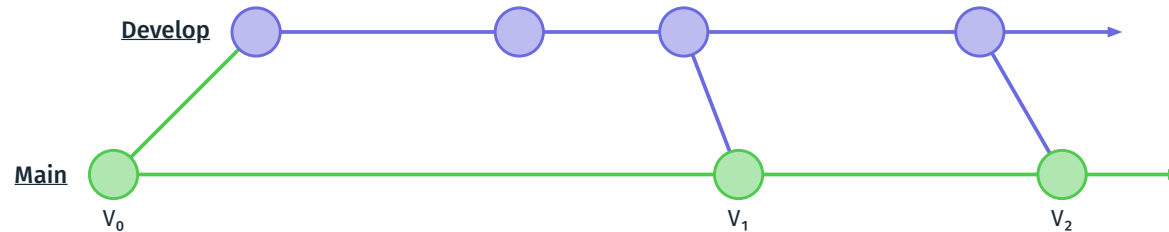
- toujours décrire ce qu'on a fait dans le message de commit :
- ne pas commit un code qui n'est pas fonctionnel
- enfin il existe une architecture typique pour les projets informatiques

# Bonnes pratiques



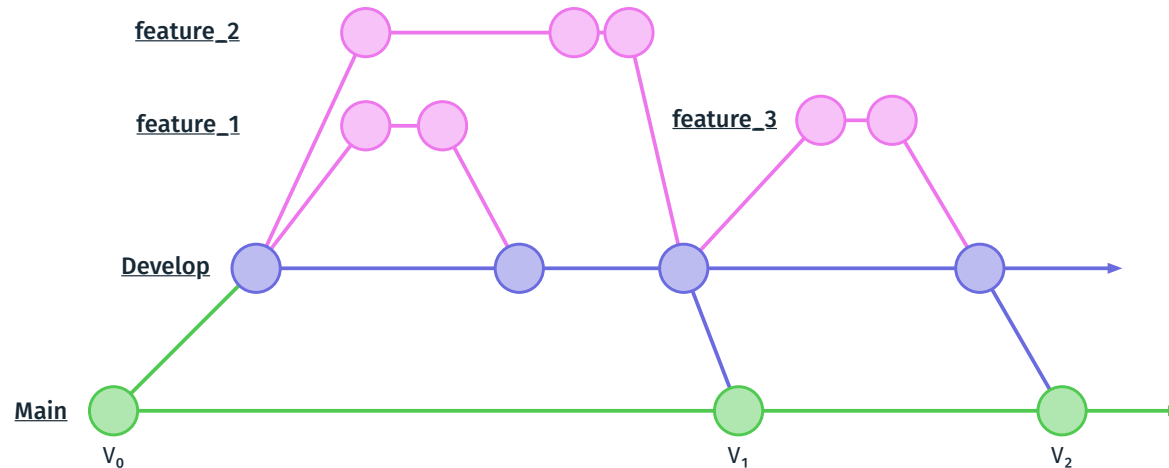
Le **main** ne reçoit que les versions complètes du projet, qui peuvent être distribuées.

# Bonnes pratiques



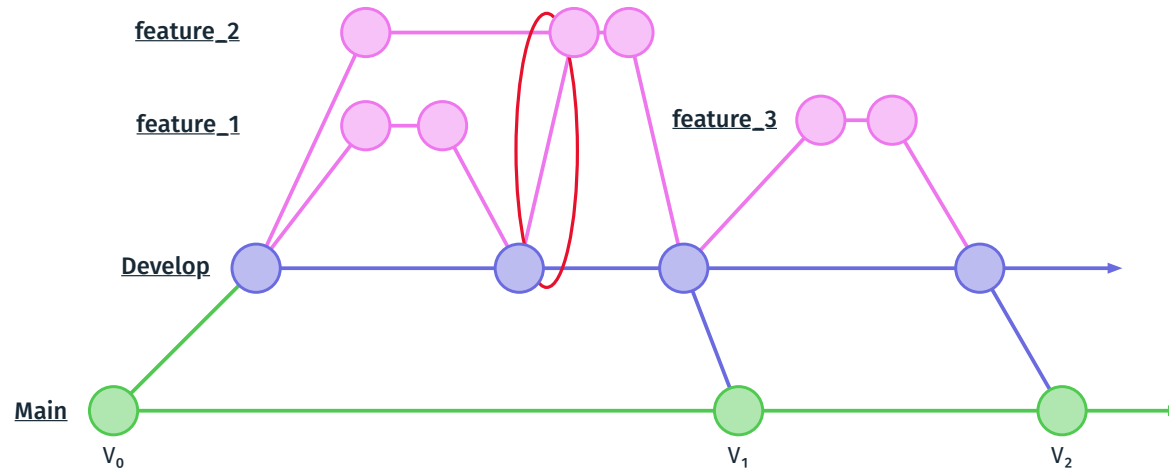
On utilise une branche **develop** pour travailler sur la création d'une nouvelle version. Cette branche doit **toujours** avoir une version fonctionnelle du projet, et on ne travaille pas directement dessus.

# Bonnes pratiques



On crée une branche pour chaque fonctionnalité à rajouter au projet. C'est sur ces branches qu'on travaille.

# Bonnes pratiques



Si la branche **develop** change, on fusionne la nouvelle version avec les branches de fonctionnalité pour **rester à jour**.

## TRAVAUX DIRIGÉS

---

**TD 2e** Publier une version  
cours2/2e\_livrer/

≈ 10'