Formation Git

Alexandre Condette - alexandre.condette@spacebel.fr

IPSA 2022 -2023 - Formation Git pour Aéro 3 et Aéro 4

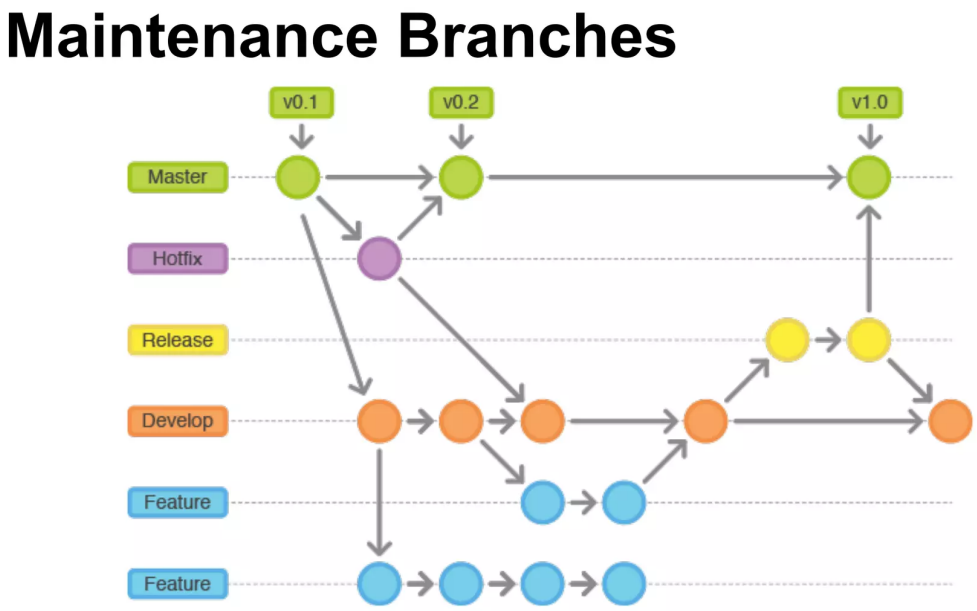
I. Prérequis

Git est un logiciel de gestion de version décentralisé.

Un logiciel de versioning, ou logiciel de gestion de version est un logiciel qui permet de conserver un historique des modifications effectuées sur un projet afin de pouvoir rapidement identifier les changements effectuées et de revenir à une ancienne version en cas de problème.

Les logiciels de gestion de versions sont quasiment incontournables aujourd'hui car ils facilitent grandement la gestion de projets et car ils permettent de travailler en équipe de manière beaucoup plus efficace.

Parmi les logiciels de gestion de versions, Git est le leader incontesté et il est donc indispensable pour tout développeur de savoir utiliser Git.



A. Télécharger et Installer Git

Avant toute manipulation, il faut vérifier que Git n'est pas déjà installé sur votre PC. Pour ce faire :

- Ouvrir un terminal

- Exécuter la commande suivante :

  ```
  git --version
  ```

  Si Git n'est pas installé, suivre la procédure ci-dessous en fonction de votre système d'exploitation

1. Pour Windows

Télécharger l'exécutable directement depuis le site de GitHub : https://gitforwindows.org/

Suivre alors les étapes d'installation. Il est conseillé pour une première installation de laisser les valeurs par défaut et de simplement cliquer sur suivant.

Une fois le processus terminé, les éléments suivants ont été installé sur la machine :

- L'outil Git
- Git bash : terminal qui permet d'utiliser Git en ligne de commande
- Gui Gui ; interface graphique qui permet de gérer les commits
- Gitk : Interface graphique qui permet de gérer l'historique d'un dépôt

2. Pour Linux

Debian / Ubuntu

```
sudo apt-get update
sudo apt-get install git
```

Fedora

```
sudo dnf install git
```

RedHat / CentOs

```
sudo yum install git
```

3. Pour MacOS

Télécharger l'exécutable directement depuis le site de GitHub : https://git-scm.com/download/mac (binary installer)

Homebrew

```
brew install git
```

MacPorts

Ouvrir un terminal et mettre à jour MacPorts

```
sudo port selfupdate
```

Rechercher les derniers ports Git disponibles et leurs variantes

```
port search git
port variants git
```

Installer Git

```
sudo port install git +bash_completion +credential_osxkeychain +doc
```

B. Créer un compte Git

Pour avoir accès aux fonctionnalités de l'interface web GitHub, il faut créer un compte en suivant la procédure de Git sur https://www.github.com/.

1. Configurer Git

Avant de commencer à utiliser Git, il est important de le configurer. Même si la configuration de base est satisfaisante, il faut configurer les informations propre à l'utilisateur, qui permettront dans l'historique de savoir qui a fait quoi.

```
# git config <option> <valeur>
git config --global user.name "Nom Utilisateur"
git config --global user.email "user@email.fr"
```

L'option --global permet d'indiquer que cette configuration est globale, et affectera tous les futurs projets.

2. Créer une clé SSH

L'utilisation d'une clé SSH avec Git permet un accès sans mot de passe aux référentiel Git. Au lieu d'identifier un utilisateur par son nom d'utilisateur et son mot de passe, la machine est authentifiée par la clé SSH.

L'utilisation de GitHub via SSH est très pratique. Une fois configurée, la clé SSH permet un accès permanent à GitHub sans aucune autre intervention.

Pour créer votre clé SSH, ouvrir un terminal et suivre la procédure ci-dessous.

```
ssh-keygen -t ed25519
```

- Appuyer 3 fois sur "Entrée" (chemin par défaut, pas de passphrase)

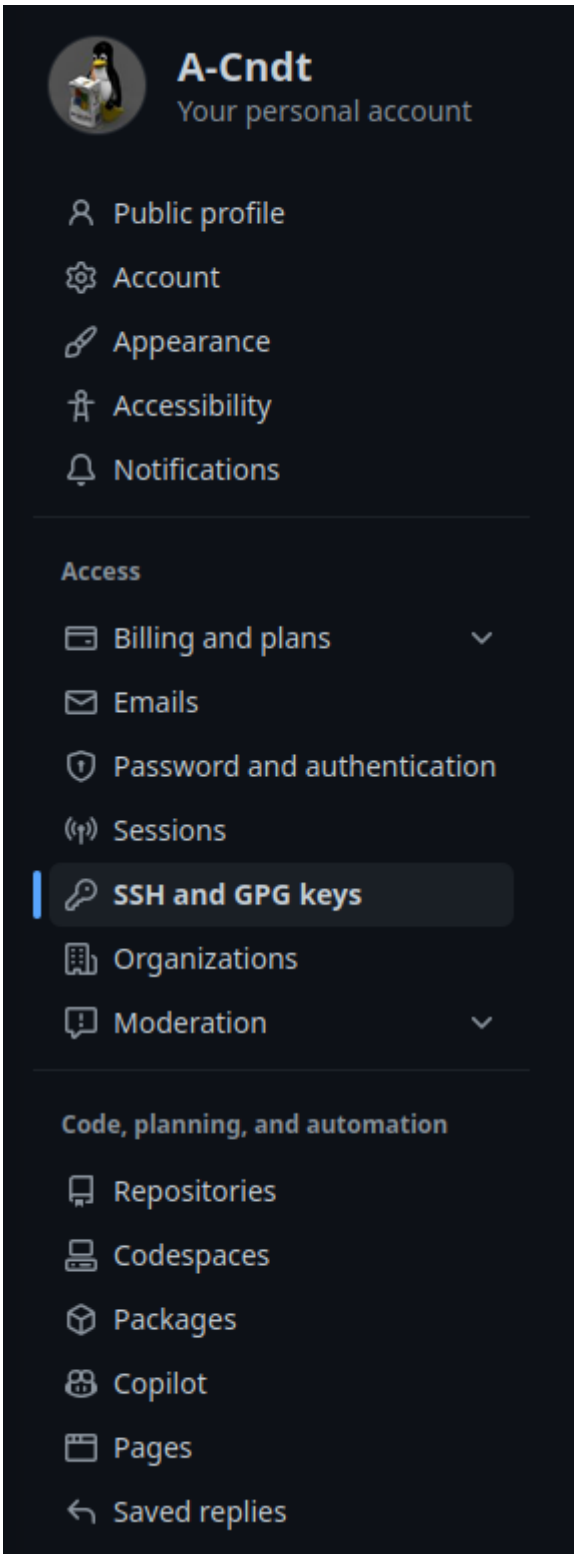
Par défaut la clé a été générée dans votre dossier :

  - C:/Users//.ssh/id_ed255919 pour Windows
  - /home//.ssh/id_ed25519 pour Linux

- Copier la clé SSH dans votre presse papier avec la commande :

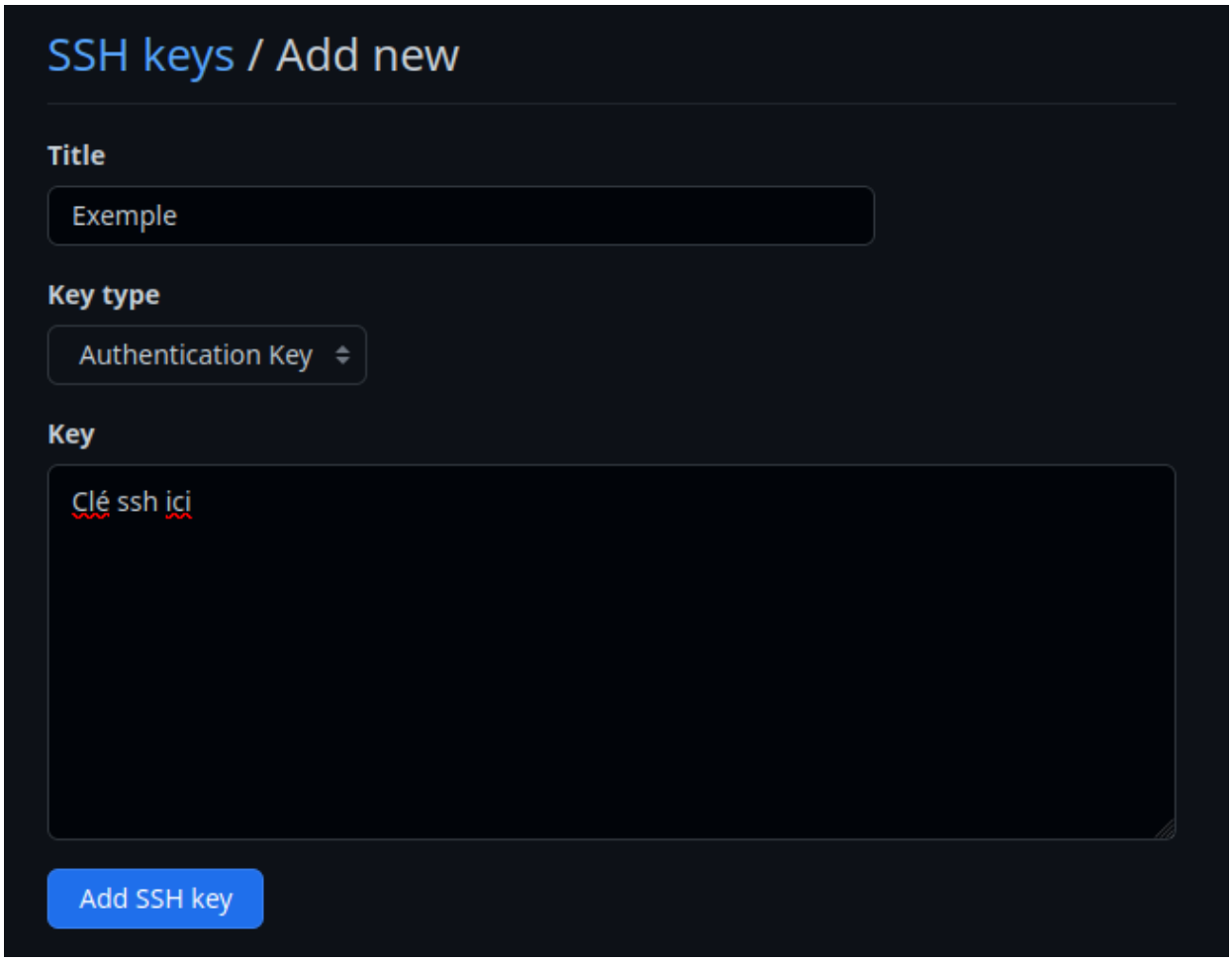
```
# Windows
clip < C:/Users/<user>/.ssh/id_ed255919.pub

# Linux
xclip  ~/.ssh/id_ed25519.pub

# Mac
pbcopy < ~/.ssh/id_rsa.pub
```

Il vous faut ensuite ajouter cette clé à votre compte github. Aller dans vos paramètres utilisateur, "sous SSH and GPG keys"



Appuyer sur **New SSH Key**, donner un nom à votre clé et coller le contenu de votre presse papier dans la textBox "Key"

Valider en appuyant sur **Add SSH key**



Vous pouvez maintenant utiliser Git

II. Initier votre premier projet avec Git

1. Créer un dépôt git à partir d'un répertoire existant

Si vous voulez convertir un projet existant sans version en dépôt Git, il vous faut vous rendre dans ce répertoire et utiliser la commande

```
git init
```

L'exécution de `git init` créer un sous répertoire *.git* dans le répertoire de travail actif. Ce dossier contient toutes les métadonnées nécessaires au dépôt Git.

2. Cloner un dépôt Git

Pour un dépôt déjà existant (par exemple un dépôt créé sous GitHub) la commande suivante permet de cibler et de créer un clone ou une copie d'un dépôt cible dans un nouveau répertoire.

```
git clone <url du depot> <nom sur votre machine>
```

Le dépôt est créé à l'endroit où est exécutée la commande et présente un environnement de travail isolé par rapport au dépôt d'origine. Autrement dit, il s'agit de votre copie de travail du projet, sur laquelle vous pouvez faire vos modification, sans affecter directement le dépôt de base (tant que vous ne commiter aucune de ces modifications)

Le clonage créer directement une connexion à distance appelée "origine" pointant sur le dépôt d'origine, pour interagir facilement avec ce dernier.

Si vous avez une clé SSH, privilégiez le clonage par SSH.

```
git clone git@github.com:A-Cndt/IPSA_Formation_GIT.git
```

Des options avancées peuvent être utilisées, par exemple pour cloner une branche spécifique, ou une version donnée de votre repo.

```
# Cloner une branche spécifique
git clone --branch <votre branche>

# Cloner un tag spécifique
git clone --branch <tag> <repo>
```

3. Bonnes pratiques

Ajouter un .gitignore

Git considère chaque fichier de votre copie de travail comme un de ces trois types :

- Tracké ; Fichier déjà stagé ou commité
- Non tracké : Fichier qui n'a pas été stagé ou commité ou un fichier ignoré
- Ignoré ; Un fichier pour lequel Git a explicitement reçu l'instruction d'ignorer

Le *.gitignore* traite ce dernier cas, pour les fichiers résultant du build des fichiers, des caches (pour python par exemple.).

Ce fichier doit être enregistré à la racine de votre repo. Il doit être édité et commité manuellement.

Voici un exemple de *.gitignore*

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so

# Distribution / packaging
.Python
build/
```

```
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
pip-wheel-metadata/
share/python-wheels/
*.egg-info/
.installed.cfg
*.egg
MANIFEST

# PyInstaller
#  Usually these files are written by a python script from a template
#  before PyInstaller builds the exe, so as to inject date/other infos into it.
*.manifest
*.spec

# Installer logs
pip-log.txt
pip-delete-this-directory.txt

# Unit test / coverage reports
htmlcov/
.tox/
.nox/
.coverage
.coverage.*
.cache
nosetests.xml
coverage.xml
*.cover
*.py,cover
.hypothesis/
.pytest_cache/

# Translations
*.mo
*.pot

# Django stuff:
*.log
local_settings.py
db.sqlite3
db.sqlite3-journal
```

```
# Flask stuff:
instance/
.webassets-cache

# Scrapy stuff:
.scrapy

# Sphinx documentation
docs/_build/

# PyBuilder
target/

# Jupyter Notebook
.ipynb_checkpoints

# IPython
profile_default/
ipython_config.py

# pyenv
.python-version

# pipenv
#   According to pypa/pipenv#598, it is recommended to include Pipfile.lock in
version control.
#   However, in case of collaboration, if having platform-specific dependencies or
dependencies
#   having no cross-platform support, pipenv may install dependencies that don't
work, or not
#   install all needed dependencies.
#Pipfile.lock

# PEP 582; used by e.g. github.com/David-OConnor/pyflow
__pypackages__/

# Celery stuff
celerybeat-schedule
celerybeat.pid

# SageMath parsed files
*.sage.py

# Environments
.env
.venv
env/
venv/
ENV/
env.bak/
venv.bak/
```

```
# Spyder project settings
.spyderproject
.spyproject

# Rope project settings
.ropeproject

# mkdocs documentation
/site

# mypy
.mypy_cache/
.dmypy.json
dmypy.json

# Pyre type checker
.pyre/
```

Ajouter un ReadMe

Un ReadMe est toujours utile dans votre référentiel pour expliquer aux autre personnes pourquoi votre projet est utile, ce qu'elles peuvent faire avec et comment elles peuvent l'utiliser

Un README est souvent le premier élément qu'un visiteur verra lors de la consultation de votre référentiel. Les fichiers README incluent généralement des informations sur :

- Ce que le projet fait
- Pourquoi le projet est utile
- Prise en main du projet par les utilisateurs
- Où les utilisateurs peuvent obtenir de l'aide sur votre projet
- Qui maintient et contribue au projet

Un README doit contenir uniquement les informations nécessaires pour que les développeurs commencent à utiliser et à contribuer à votre projet. Les documentations plus longues sont mieux adaptées aux wikis.

Le ReadMe est rédigé en Markdown

Les choses à faire quand vous travaillez en équipe sur un repo

La meilleure façon d'éviter les conflits sur un repo Git lorsque l'on travaille en équipe sur un projet sont de respecter les règles suivantes :

- Éviter de modifier le même fichier en même temps qu'un autre développeur.
- Avant chaque modification faire un `git pull` pour récupérer les modifications des autres.
- `git commit` et `git push` régulièrement
- Faire des `git rebase` si besoin

- Avoir un "Expert Git" qui gère le repo, et qui soit le seul à pouvoir merger sur la branche principale

III. Premiers Commits

1. Git status

La commande status permet d'obtenir un statut sur l'état actuel de votre versionning. Elle donne un aperçu des fichiers qui ont été modifiés, ceux qui sont en stagging et ceux qui ne sont pas suivis

```
git status
```

Cette commande est à utiliser le plus souvent possible pour savoir où vous en êtes et éviter de mauvaise surprises par la suite.

```
ace@pcAlex:~/Bureau/IPSA_Formation_GIT$ git status
Sur la branche main
Votre branche est à jour avec 'origin/main'.

rien à valider, la copie de travail est propre
```

```
ace@pcAlex:~/Bureau/IPSA_Formation_GIT$ git status
Sur la branche main
Votre branche est à jour avec 'origin/main'.

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
        Exemple.txt

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents (utilisez "git add" pour les
 suivre)
```

2. Git Pull

La commande `git pull` est utilisée pour faire un fetch du contenu d'un dépôt distant pour le télécharger, et ainsi mettre à jour directement le dépôt local.

En réalité, le `git pull` est une combinaison d'un `git fetch` suivie d'un `git merge`. `Git fetch` télécharge le contenu du dépôt distant, ensuite `git merge` fait une fusion des heads et contenus dans un nouveau commit de merge local.

```
git pull <remote>

# Sans commit de merge
git pull --no-comit <remote>

# Pull verbeux
git pull --verbose
```

3. Git Add

Le `git add` ajoute un changement dans le répertoire de travail à la zone de stagging. Celà informe Git que vous voulez inclure des mises à jour dans un fichier en particulier du commit suivant.

Un add n'affecte pas le dépôt directement, les changements ne sont enregistrés que lors du commit.

Il est souvent judicieux de faire un git status après un add pour être certains de commiter les bonnes modifications.

La zone de stagging est une zone "tampon" entre le répertoire de travail et l'historique du projet. Au lieu de commiter tous les changements apportés depuis le dernier commit, a zone de staging vous permet de regrouper les changements liés dans des instantanés ciblés avant de les commiter dans l'historique du projet. Autrement dit, vous pouvez apporter différents changements à des fichiers distincts, revenir en arrière et les séparer en commits logiques en ajoutant des changements liés au staging, puis les commiter un par un.

```
# Ajouter un fichier
git add <file>

# Ajouter un dossier
git add <directory>

# Ajouter tous les changements (non recommandé la plupart du temps)
git add *
```

4. Git Commit

Le commit capture un instantané des changements actuellement staggé (donc après un `git add`) du projet.

Les instantanés doivent être des version "sûres" du projet, c'est à dire une version fonctionnelle.

Chaque commit doit être le plus atomique possible (Donc des commits fréquents, par petites unités pour chaque fonctionnalité afin de pouvoir facilement identifier le commit en cause (et donc le code défaillant) lors de l'apparition de bugs) quitte à ensuite regrouper les commits connexes et de nettoyer l'historique local avant un merge vers la branche principale (via les fonctions avancées)

Il est toujours préférable d'ajouter un ou plusieurs commentaires à son commit afin d'expliquer brièvement à quoi sert ce commit, ainsi que le cas échéant, à quelle issue, demande de feature il fait référence. Celà permettra dans GitHub de suivre tous les commits liés à un bug / une feature.

```
# Commit qui référence la feature #1
git commit -m "Ceci est un commit d'exemple (refs #1)"

# Commit avec multiple commentaires
git commit -m "Commentaire principal (refs #1)" -m "Commentaire secondaire qui peut
être plus détaillé"
```

On peut également modifier le dernier commit tant qu'il n'a pas été push, pour ajouter des fichiers à ceux déjà staggés

```
git commit --amend
```

5. Git Push

La commande `git push` est utilisée pour charger le contenu d'un dépôt local vers un dépôt distant. Le push vous permet de transférer les commits de votre dépôt local vers un dépôt distant. C'est l'équivalent de `git fetch`, mais à l'inverse du fetch qui importe les commits dans des branches locales, le push les exporte vers des branches distantes. Ces dernières sont configurées à l'aide de la commande `git remote`. Le push est susceptible d'écraser les changements. Vous devez donc prendre des précautions lorsque vous l'exécutez.

```
# Push d'une branche spécifique
git push <remote> <branch>
```

IV. Revenir en Arrière

1. Git log

La commande `git log` permets d'afficher un historique des commits

```
git log <option> <revision-range> [--] <path>
```

Les différentes options sont trouvables dans la doc de git et permettent par exemple de limiter le nombre de commit à afficher.

2. Git diff

Le diff prend 2 ensembles de données et génère une sortie révélant les changements entre eux. Les sources de données peuvent être des commits, des branches, des fichiers etc.

On utilise souvent le diff avec les commande `git status` et `git log` pour analyser l'état d'un dépôt

```
git diff <source> <fichier à comparer>
```

Git diff possède un mode spécial pour mettre en évidence les changements avec une meilleure granularité

```
git diff --color-words
```

3. Git checkout

La commande checkout à plusieurs utilité :

```
# Créer une nouvelle branche
git checkout -b <new-branch>

# Basculement sur une branche existante
git checkout <branch>
```

Mais en utilisant git log pour obtenir l'identifiant d'un commit précédent par exemple le commit *435b61d - Initial Commit*, vous pouvez faire un checkout de ce commit pour rétablir le dépôt à l'état dans lequel il était lors de ce commit. Le mieux pour travailler dans cette version antérieure et de le faire dans une nouvelle branche car cette commande nous place dans un etat particulier appelé "detached HEAD", on est revenu en arrière en tant que spectateur, et on peut voir le projet tel qu'il était à l'époque du commit, tout en ayant la possibilité de revenir dans le présent.

4. Git revert

Cette commande permet d'inverser un commit, elle va défaire ce qui a été fait au moment du commit en créant un nouveau commit. Celà n'altère pas l'historique mais ajoute un nouveau commit d'inversion.

```
git revert <commit>
```

5. Git Reset

`git reset` comme `git checkout` permet de faire plusieurs choses à la fois. Mais elle altère l'historique et peut supprimer vous modification.

```
# Supprime un fichier de la zone de stagging sans altèrer les modifications
git reset <fichier>

# Revenir en arrière jusqu'a un commit donnée
git reset <commit>
```

La commande `git reset <commit>` réinitialise la zone de staging, mais laisse votre fichier de travail en l'état. En revanche l'historique des commits suivant celui donnée en argument sera perdu

La méthode pour tout réinitialiser comme à l'état du commit (donc la zone de staging, et le répertoire de travail) est :

```
git reset <commit> --hard
# Cette commande est à utiliser avec précautions
```

V. Fonctions avancées

1. Git merge

La fonction de merge permet de ramener une branche sur une autre et les fusionne. La fusion de 2 branches se fait toujours à partir de la branche principale

- La branche source sera affectée en récupérant l'historique de la branche.
- La branche fusionnée ne sera pas affectée

```
# Fusionner la branche <branche> avec la branche courante
git merge <branche>
```

2. Git rebase

Ce commande permet de déplacer une branche et de changer son commit de départ. Cela permet de "rapatrier" les commits d'une autre branche et celà doit être fait régulièrement pour que chaque membre du projet garde une version à jour de chaque fichier. Le rebase permets notamment de récupérer les changements faits sur un fichier que vous êtes vous même en train d'éditer.

```
git rebase <nouvelle-base>
```

On peut également utiliser l'option -i pour déclencher un rebase interactif, et qui permets par exemple de réorganiser les commits, des les "squash" (fusionner 2 commits par exemple pour 2 commits sur une même feature)

Avec `git rabse -i` on a les options suivantes :

- pick : inclure le commit, On peut en profiter pour changer l'ordre des commits
- reword : modifier le message de commit
- edit : permet d'éditer le commit (par exemple le séparer en plusieurs sous-commit si on a fait un commit trop important)
- squash : combine le commit avec celui qui le précède
- fixup : comme squash mais utilise le message du commit précédent
- exec : executer une commande shell sur le commit

```
# Modifier les 6 derniers commits
git rebase -i HEAD~6
```

3. Git stash

`git stash` est une fonction de remisage. Les contraintes d'un projet, la répartition des rôles dans ce dernier nous oblige parfois à arrêter le travail en cours et de passer sur une autre branche pour travailler sur une autre feature Si l'état de notre répertoire de travail lors du changement de branche ne permets pas de faire un commit propre (code non fonctionnel par exemple, conflits de merge) il faut utiliser le système de remisage pour stocker temporairement les modifications effectuer afin de les reprendre plus tard

```
# Mettre de cote les modification en cours
git stash

# Reappliquer les changements
git stash apply

# Voir la liste des stash
git stash list

# Annuler le dernier stash
git stash drop
```