TP2-Partie 2 : Gestion de branches et de conflits

Dans ce TP, nous allons aborder les notions suivantes :
• La création et le changement de branches.
• L'intégration des modifications, c'est-à-dire la fusion de branches.
• La réécriture d'historique.
• La gestion de conflits de fusion.

Partie 4 : Rebase interactif

Placez-vous de nouveau dans un répertoire spécifique à cette partie du TP.

a. Récupérez une copie locale du dépôt grâce à la commande suivante : git clone https://gitlab.com/git-course-mpoquet/exercise-branch1.git
b. Nous allons ici montrer la puissance de réécriture d'un rebase interactif, en montrant des opérations courantes qu'il permet. Assurez-vous d'être dans master et tapez git rebase -i --root, qui permet de réécrire l'historique de manière interactive (-i) depuis le début de dépôt (--root) jusqu'à HEAD. Cette commande va ouvrir votre éditeur de texte afin de vous demander ce que vous souhaitez faire de chaque commit. Les commits sont triés par ordre dans lequel ils vont être réécrits. Comme indiqué en commentaire du fichier ouvert dans votre éditeur, vous pouvez faire différentes actions sur chaque commit. Ici, nous allons faire les modifications suivantes :

   - Simplifier le fichier de licence dans le commit initial ac1345c.
   - Mettre "doc: initial README" en message de commit de 74d41b0.
   - Fusionner les commits 882c921 et 0aaccd9 en un seul commit. Cela se fait via un squash sur le dernier des deux commits (0aaccd9)

   Ce qui devrait se traduire par ce contenu dans votre éditeur de texte :
   - edit ac1345c initial commit
   - reword 74d41b0 doc: README
   - pick 882c921 code: first version
   - squash 0aaccd9 code: fix shabang
c. Après avoir enregistré le fichier et quitté votre éditeur de texte, le rebase interactif commence. À chaque étape, git rebase devrait vous dire ce que vous êtes en train de faire et ce qu'il reste à effectuer. Notez que git rebase --abort permet d'annuler toute l'opération de rebase, ce qui est très utile en cas de panique.
   1. Tout d'abord, il devrait lancer l'édition du commit initial. Il va pour cela vous replacer dans votre terminal dans un état spécial de rebase interactif — lancez git status pour voir l'état dans lequel vous êtes. Puisque vous êtes en mode édition d'un commit, vous pouvez librement modifier des fichiers et ajouter ces modifications au commit. Ne gardez que la première et la dernière ligne du fichier UNLICENSE et sauvegardez le fichier. Référez-vous à git status pour

savoir comment enregistrer votre modification puis pour savoir comment continuer vos opérations de rebase « git rebase –continue ». (Git va probablement appeller à ce moment-là votre éditeur pour vous proposer de modifier le message de commit. Dans ce cas, laissez le contenu inchangé, enregistrez le fichier et fermez votre éditeur de texte.)

2. Ensuite, il devrait appeler votre éditeur de texte pour changer le message de commit de 74d41b0. Écrivez donc “doc: initial README” puis enregistrez et quittez le fichier.
3. Il devrait ensuite appeler votre éditeur de texte pour choisir un message de commit pour les deux commits à fusionner. Vous pouvez garder seulement le message de commit du premier fichier, pour faire comme si on avait directement écrit ce fichier avec le bon shabang du premier coup. Supprimez donc les autres lignes non commentées du fichier, enregistrez-le et quittez votre éditeur de texte.

Une fois le rebase interactif terminé, jetez un œil à votre historique Git pour vérifier qu’il comprend bien les modifications souhaitées.

Partie 5 : Comment gérer un conflit ?

Normalement, les appels précédents dans ce TP de git merge et git rebase n’ont pas créé de conflits. Dans la pratique, des conflits arrivent lorsque plusieurs modifications ont été faites sur les mêmes bouts de code, ce qui arrive relativement fréquemment lorsqu’on travaille à plusieurs en parallèle sur un projet. Nous allons voir les deux manières principales de gérer un conflit en Git.

a. Récupérez une copie locale du dépôt qui va faire apparaître des conflits en lançant git clone https://gitlab.com/git-course-mpoquet/exercise-branch2.git.
b. Observer le dépôt : jetez un œil à son historique et au contenu des différents fichoers dans chaque branche avec la commande git diff Nom_branche_1 Nom_branche_2
c. Essayons maintenant de fusionner function-politeness dans master grâce à git merge. La commande devrait échouer et vous dire qu’un conflit a été détecté, en plus de vous dire sommairement comment le résoudre. Lancez git status pour voir quelles commandes effectuer en détails, en plus de vous donner la commande pour annuler la tentative de fusion et revenir dans l’état précédent.

Partie 6 : Résoudre le conflit directement

Résoudre un conflit est tout à fait faisable sans autre outil qu’un éditeur de texte en Git. Lorsque un conflit est détecté, Git vous fait passer en mode résolution de conflits, qui est très proche du mode édition de commit que l’on vient de voir avec un rebase interactif.
Par défaut, Git va modifier les fichiers qui ont des conflits pour y faire apparaître les lignes conflictuelles, entourées de guides permettant de voir quelles lignes ont été modifiées dans quelle branche. Ici, un conflit est apparu sur le fichier hello.py, affichez donc son contenu. La partie conflictuelle devrait ressembler
aux lignes suivantes :

```
<<<<<<< HEAD
print(f'Hello, {stuff}!')
```

=======
print(f'Hi {someone}')
>>>>>>> function-politeness

Cet affichage indique que votre version courante (HEAD) du fichier utilise print(f'Hello, {stuff}!') tandis que celle de la branche function-politeness utilise print(f'Hi {someone}'). Ces informations, avec l'aide du code autour du conflit, vous permettent un peu de comprendre ce qui s'est passé et d'éventuellement résoudre le conflit. Cependant, il manque ici une information qui peut être cruciale pour savoir rapidement comment résoudre un conflit sans parcourir l'historique Git : quel était le contenu des lignes conflictuelles dans la base commune de ces deux branches ? Git permet d'afficher cette information, mais il ne le fait pas par défaut.
Recommençons la résolution de conflit en affichant cette information manquante.
Annulez la fusion en cours (git status devrait vous dit comment faire), puis configurez le style de conflit de fusion en diff3 grâce à la commande suivante : git config --global merge.conflictstyle diff3.
Recommencez ensuite la fusion, qui devrait échouer comme précédemment. Le fichier hello.py devrait par contre avoir changé et contenir les lignes suivantes :
<<<<<<< HEAD
print(f'Hello, {stuff}!')
||||||| merged common ancestors
print(f'Hi {stuff}')
=======
print(f'Hi {someone}')
>>>>>>> function-politeness

L'information ajoutée est ici importante, puisqu'on voit en un coup d'œil quelle modification a été faite dans HEAD et quelle modification a été fait dans function-politeness, pas uniquement le contenu final des lignes dans chaque branche. Dans HEAD le mot Hi a été remplacé en Hello, et un ! final a été ajouté, alors que dans function-politeness la variable stuff a juste été renommée en someone. Une manière efficace de résoudre le conflit est donc de prendre la version de HEAD pour conserver son format, en remplaçant la variable stuff par someone. Modifiez le fichier hello.py pour faire cette modification, en supprimant également toutes les lignes de guide (<<<, |||, === et >>>) afin que le fichier contienne du Python valide — vérifiez que le fichier hello.py marche comme prévu quand vous l'exécutez. Sauvegardez le fichier puis finissez la fusion via des commandes Git (git status vous dit comment faire).

Partie 7 : Utiliser un outil de résolution de conflit

Les conflits Git sont souvent résolus directement à la main comme dans la section précédente. Git permet aussi d'utiliser un outil de résolution de conflit à la place, ce que nous allons voir maintenant.
Nous allons ici nous servir de kdiff3, un des seuls outils libres de gestion de conflit permettant d'afficher la version de base de fichier. Tout d'abord, installez kdiff3 s'il n'est pas déjà accessible sur votre machine.

```
git config --global merge.tool kdiff3
git config --global mergetool.kdiff3.path "C:/ProgramFiles/KDiff3/kdiff3.exe"
git config --global mergetool.kdiff3.trustExitCode false

git config --global diff.guitool kdiff3
git config --global difftool.kdiff3.path "C:/Program Files/KDiff3/kdiff3.exe"
git config --global difftool.kdiff3.trustExitCode false
```

Configurez ensuite Git pour lui dire qu'il doit se servir de kdiff3 comme outil de résolution de conflits via la commande git config --global merge.tool kdiff3

a. Repartez d'une copie fraiche de dépôt précédent grâce à la commande git clone https://gitlab.com/git-course-mpoquet/exercise-branch2.git, puis tentez de nouveau de fusionner function-politeness dans master, ce qui devrait encore échouer.
b. Lancez maintenant git mergetool, qui va vous lancer l'interface graphique de kdiff3. Cette interface vous affiche simultanément les différentes versions du fichier (la version de base, la version courante et celle à fusionner), en colorant les différences au sein de chaque ligne de chaque version par rapport à la version de base. Le dernier fichier (en bas) est le fichier résultant de la résolution de conflits.

Vous pouvez résoudre le conflit dans kdiff3 en faisant un clic droit sur les lignes conflictuelles restantes pour lui dire quelle version prendre, puis éditer cette ligne à la main si nécessaire. Si vous jetez un œil aux fichiers de votre dépôt pendant que le mergetool s'exécute, vous pouvez voir que les différentes versions du fichier ont été copiées dans des fichiers temporaires, ce qui peut être pratique pour faire différentes opérations sur des fichiers (comme appeler diff).
Résolvez le conflit dans kdiff3, vérifiez que hello.py s'exécute comme prévu, puis finissez la fusion via des commandes Git (git status est encore là pour vous aider)