Feuille d'exercices : initiation à Git

0. Setup

- 0.1 : Vérifiez que vous avez accès à la machine de travail fournie par le formateur avec le login `stagiareX` (`X` entre 1 et 10) et comme mot de passe `ilovegit`
- 0.2 : Une fois connecté, ouvrez un terminal et vérifiez que `git` est installé à l'aide de `git --version`
- 0.3 : Vérifiez que votre IDE préféré est installé

1. Les bases : Dépôt, commit, état des fichiers

- 1.1 : Récupérer les fichiers `multiplication.py` et `main.py` auprès du formateur. Les mettre dans un nouveau dossier `multiplication` .
- 1.2 : Observez ce que fait ce programme : déplacez-vous dans le dossier `multiplication` et lancez `python main.py`
- 1.3 : Initialisez un nouveau dépôt Git à l'aide de `git init` puis `git status` pour confirmer que le dépôt est bien initialisé. Vous pouvez aussi contaster qu'il existe maintenant un dossier `.git` .
- 1.4 : Avant d'effectuer votre premier commit, vous aurez sans doute besoin d'initialiser votre identité Git sur le système :

```
git config --global user.name "<votre nom>"
git config --global user.email "<votre email>"
```

Au passage, on peut dire à Git d'utilisez des couleurs dans `git status` et ailleurs :

```
git config --global color.ui true
```

Constatez que ces informations ont été sauvegardées dans le fichier `.gitconfig` dans votre répertoire personnel.

- 1.5 : Ajoutez les fichiers avec `git add *` et faites un premier commit avec `git commit` . Généralement le commentaire pour le tout premier commit est quelque chose comme `Initial commit` . Vérifiez ensuite l'état du dépot et de l'historique avec `git status` et `git log` .
- 1.6 : Oups ! Vous réalisez que vous avez versionné le dossier `__pycache__` qui n'a pas vocation à être versionné. Demandez à Git de l'oublier avec `git rm -r __pycache__` (attention, cette commande a aussi pour effet de réellement supprimer le dossier de l'espace de travail actuel), puis de nouveau `git commit` .
- 1.7 : Créez un fichier `.gitignore` pour ignorer le dossier `__pycache__` qui n'a pas vocation à être versionné. Commitez ensuite ce nouveau changement. Vérifiez avec `git status` que `__pycache__` n'est plus listé comme "fichier non versionné".
- 1.8 : Arrivez-vous à créer un dossier *vide* qui s'appelerait `resultats` et à le versionner avec Git ?

2. Explorer un dépôt Git

- 2.1 : Clonez le dépôt "microblog" indiqué par le formateur
- 2.2 : Installez ce qui est nécessaire pour l'application avec les commandes suivantes:

```
sudo apt install python3-pip python3-venv
cd dossier/de/travail
python3 -m venv venv
source venv/bin/activate
pip3 install -r requirements.txt
flask db init
flask db upgrade
```

- 2.3 : Lancez l'application avec `flask run` vous devriez voir qu'elle écoute sur le port 5000
- 2.4 : Depuis un navigateur sur la machine, accédez à `http://localhost:5000/` . Créez vous un compte et postez un message.
- 2.5 : Depuis la partie "Profile", tentez d'exporter vos messages.
- 2.6 : Plutôt que d'utiliser la version finale de l'application, remontons l'historique du dépôt pour retrouver un état de l'application sans cette fonctionnalité buggé. En utilisant `git blame` sur le fichier `app/main/routes.py` , arrivez-vous à trouver le commit qui a introduit la fonctionnalité d'export ?
- 2.7 : Même question mais en utilisant votre IDE préféré. En particulier si vous utilisez VScode, installez les extensions GitLens et Git Graph.
- 2.8 : Placez-vous sur le commit avant l'introduction de cette fonctionnalité, puis relancez l'application. Confirmez-vous que la fonctionnalité a disparu depuis votre navigateur ?
- 2.9 : Remettez-vous sur le commit initial, puis refaite la même manipulation depuis VScode / Eclipse (ou vice-versa depuis le terminal, si vous étiez déjà passé par l'IDE)

3. Les branches

- 3.1 : Identifiez les noms de branche et de tag dans l'historique à l'aide de `git log --oneline` , `tig` ou VScode / Eclipse
- 3.2 : Retournez à la fin de l'historique à l'air de `git checkout main`
- 3.3 : En reprenant le commit identifié à la question 2.6, nous allons réinitialiser violemment l'historique du projet avec `git reset --hard <commit_id>` . Que constatez-vous dans `git status` et `git log` . En particulier, vers quel commit pointe maintenant la branche `main` ? *NB: utiliser `git reset --hard` est une manipulation qui a des impacts importants, et doit être utilisé avec précaution. En tout cas, cette manipulation est juste proposée ici à titre d'illustration pédagogique et n'a pas de rapport avec les énoncés suivants !*
- 3.4 : On se propose maintenant de créer une branche pour étendre l'application avec une page supplémentaire “A propos”. Pour ce faire, commencez par créer une branche nommée `about-page` et vous positionner dessus.
- 3.5 : Trouvez comment ajouter une nouvelle page "A propos" dans l'application. Il vous faudra ajouter un controlleur dans `app/main/routes.py` , un template dans `app/templates/about.html` , et un nouveau lien dans `app/templates/base.html` . Par exemple:

```
### Dans routes.py

@bp.route('/about')
```

```
def about():
    return render_template('about.html')
```

```
<!-- Dans about.html -->

{% extends "base.html" %}

{% block app_content %}
<h1>About</h1>

This is a simple microblogging app
{% endblock %}
```

```
<!-- Dans base.html (à l'endroit approprié) -->

<li><a href="{{ url_for('main.about') }}">{{ _('About') }}</a></li>
```

- 3.6 : Commitez l'ensemble de ces changements (n'oubliez pas d'ajouter les nouveaux fichiers non-versionnés avec `git add` si besoin !)
- 3.7 : Utilisez `git reset HEAD~1` pour faire un "soft" reset qui annule votre dernier commit (mais conserve les fichiers dans l'état actuel, à la différence du `git reset --hard`). Puis refaites ce commit depuis VS code / Eclipse.
- 3.8 : Utilisez `git reflog` pour relire l'historique de tout vos changements de commit / état du dépôt

4. Les remotes, les merges, les merge-request

- 4.1 : Créez-vous un compte sur https://gitlab.com (pas la peine d'utilisez votre vrai nom ni une vraie adresse-mail)
- 4.2 : Rendez-vous sur le dépôt original de microblog, puis forkez le projet à l'aide du bouton en haut à droite de la page
- 4.3 : Ajoutez ce nouveau remote dans votre clone local à l'aide de `git remote add`
- 4.4 : Poussez votre branche `about-page` sur votre fork
- 4.5 : Confirmez que vous trouvez bien cette nouvelle branche sur votre fork depuis votre navigateur, puis allez dans la partie "Merge request". Créez une nouvelle "merge request" en prenant bien soin de sélectionner la branche du formateur (sur le dépôt original !) comme cible.
- 4.6 : Vérifiez que la merge request a bien été crée sur le dépôt du formateur et est en attente de relecture/validation
- 4.7 : Pendant ce temps, le formateur continue de travailler sur sa branche `main` et va bientôt commiter un changement qui va créer un conflit entre `main` et votre branche (attendre le signal du formateur ;)). Une fois que c'est fait, vous devriez voir sur la page de la merge request qu'une vérification de mergeabilité effectuée par GitLab est passée au rouge.
- 4.8 : Utilisez `git pull` (ou bien `git fetch` et `git merge` séparément) pour fusionner la branche du formateur dans la votre, et résolvez le conflit. Poussez ensuite le nouveau commit sur votre branche et validez que la vérification de Gitlab est repassée au vert.
- 4.9 : Le formateur devrait également avoir laissé une petite revue de code contenant une suggestion de changement. Utilisez l'interface de GitLab pour transformer cette suggestion en

commit, et synchronisez de nouveau votre branche locale. Vérifiez que la suggestion du formateur est bien présente dans la sortie de `git log` ou dans Git Graph de VSCode.

- 4.10 : Une fois que le formateur a mergé votre merge-request (ou celle d'un.e camarade !), re-synchronisez votre dépôt local ainsi que votre fork.

5. Bonnes pratiques, situations de la vie quotidienne

- 5.1 : Toujours dans le dépôt `microblog`, de retour sur `main`. Nous allons **tester la commande `git stash`**. Pour cela nous allons simuler une situation où nous nous apprêtons à `git pull` des commits en ayant des changements non commités.
  - Revenez en arrière dans l'historique avec `git reset --hard v0.15`
  - Modifiez un fichier, par exemple `requirements.txt`, en ajoutant des commentaires à la fin ou au début
  - Lancez `git pull` : Git refuse car le merge ne peut pas avoir lieu tant que vous avez des changements non commités
  - Mettez de côté temporairement vos changements non commités avec `git stash` (vérifier le résultat avec `git status`)
  - Ré-effectuez le `git pull` qui devrait fonctionner
  - Ré-appliquez vos changements non commités avec `git stash pop` (vérifier le résultat avec `git status` et `git diff`)
- 5.2 : Toujours dans le dépôt `microblog`, de retour sur `main`. Nous allons **effectuer un commit que nous aurions voulu en fait séparer en plusieurs commit distincts**.
  - modifier un ou plusieurs fichiers de sortes à avoir au moins deux changements différents
  - commitez ces changements dans un seul commit
  - ... oups ! Nous aurions voulu faire plusieurs commit :) ...
  - pour "annuler" notre dernier commit mais sans perdre nos modification, nous utilisons `git reset HEAD~`. Confirmez avec `git log` que le commit n'est plus là, mais que `git diff` montre que nos modifications n'ont pas été perdues
  - faites un premier commit qui commitera seulement l'un des deux changements
  - faites un deuxième commit avec le changement restant
- 5.3 : Toujours dans le dépôt `microblog`, de retour sur `main`. Nous allons **effectuer un commit sur `main`, que nous aurions voulu en fait mettre sur une nouvelle branche**.
  - Modifiez quelques fichiers et commitez sur `main`
  - ... oups ! Nous aurions voulu commiter sur une nouvelle branche
  - pour "annuler" notre dernier commit mais sans perdre nos modification, nous utilisons `git reset HEAD~`. Confirmez avec `git log` que le commit n'est plus là, mais que `git diff` montre que nos modifications n'ont pas été perdues (tiens donc, tout cela ressemble furieusement à l'exercice précédent !)
  - crééz et passez sur une nouvelle branche avec `git switch -c <votre_branche>`
  - commitez le changement sur la branche.
- 5.4 : Récupérez auprès du formateur un petit fichier de patch contenant la sortie d'un `git diff`. Appliquez ce patch sur votre espace de travail en lançant `git apply`. Cette commande tourne "dans le vide" en attendant que vous colliez le contenu du patch, puis que vous fassiez Ctrl+D pour terminer. Vérifiez avec `git status` et `git diff` que le patch a bien été appliqué sur votre espace de travail.

6. Configurer Gitlab CI

- 6.1 : Depuis `main`, créez une nouvelle branche `ajout-test-ci`. Dans cette branche, rajoutez via un *cherry-pick* le commit que vous aurez trouver dans sa branche (sur `origin`) qui rajoute un le fichier `.gitlab-ci.yml` contenant une sorte de Hello-World.

- 6.2 : Rajoutez un nouveau job qui va installer l'outil `flake8` et le lancer sur le fichier `app/main/routes.py` :

```
apt update
apt install python3-pip -y
pip3 install flake8
flake8 app/main/routes.py
```

- 6.3 : Commitez vos changements, poussez votre branche sur votre fork, et créez une nouvelle pull-request. Constatez également que, normalement, la pipeline s'est déclenchée pour faire tourner le test.
- 6.4 : Corrigez le fichier `app/main/routes.py` pour que `flake8` soit content
- 6.5 : Finalement, testons le fonctionnement de `git rebase`
  - Re-créez une toute nouvelle branche `superbranche` qui commencera depuis le tag `v0.21`
  - Utilisez des `git cherry-pick` pour ajouter votre (ou vos) commits qui rajoutait la page "About"
  - De même pour les commits qui rajoutaient la CI dans Gitlab
  - Regardez la structure actuelle des différentes branche dans VScode (ou avec `git log --oneline --graph`)
  - `rebase` -ons votre branche de sorte à ce qu'elle démarre depuis le sommet de la branche `main` , en utilisant `git rebase main`
  - Comparez la nouvelle structure de branche