TP — Comprendre et utiliser `.gitignore`

Objectifs

- expliquer le rôle d'un fichier `.gitignore`
- constater que `.gitignore` n'agit pas rétroactivement
- corriger un dépôt Git "pollué"
- utiliser Git (ou VS Code) pour retirer des fichiers du suivi

Contexte

Un projet Python vous est fourni sous la forme d'un **modèle GitHub (template)**.

À partir de ce modèle, vous avez créé **votre propre dépôt GitHub**, puis vous l'avez **cloné sur votre machine**.

Ce dépôt contient volontairement :

- un environnement virtuel Python (`.venv` sur Linux/macOS ou `venv` sur Windows)
- des fichiers générés automatiquement (`__pycache__`, `*.pyc`)
- **aucun fichier `.gitignore` au départ**

> Ces fichiers **ne devraient pas être versionnés**, mais ils sont présents **volontairement** pour les besoins du TP.

> *Ce TP volontairement "mal parti" simule une situation fréquente : travailler sur un dépôt Git existant qui n'a pas été créé avec les bonnes pratiques.*

Remarques

> `.gitignore` empêche Git de suivre de *nouveaux* fichiers, mais **ne supprime jamais ceux déjà enregistrés**.

Étape 0 — Préparer votre dépôt

Étape 0.1 — Créer votre dépôt de TP

1. Ouvrez le lien du dépôt fourni
2. Cliquez sur **Use this template**

   > Ne forkez pas le dépôt. Utilisez **Use this template** uniquement.

3. Nommez votre dépôt : `tp-gitignore-ciel`
4. Créez le dépôt sur votre compte GitHub

Étape 0.2 — Cloner votre dépôt

```
git clone https://github.com/<votre-compte>/tp-gitignore-ciel.git
cd tp-gitignore-ciel
```

> À partir de maintenant, **vous travaillez sur votre propre dépôt**.

Étape 0.3 — Activer l'environnement virtuel

Windows (PowerShell ou CMD)

```
.venv\Scripts\activate
```

> Si PowerShell bloque l'exécution, vous pouvez temporairement autoriser :
>
> ```
> Set-Ex
> ```

Linux / macOS

```
source .venv/bin/activate
```

> Après activation, vous pouvez installer les dépendances du projet (voir `requirements.txt`) :

```
pip install -r requirements.txt
```

Vérifier que tout est installé

```
pip list
```

Vous devriez voir toutes les librairies nécessaires au projet :

```
Package         Version
--------------- -----------
colorama        0.4.6
pip             24.2
python-dateutil 2.9.0.post0
six             1.17.0
tabulate        0.9.0
```

Étape 1 — Observation d'une erreur courante

Travail demandé

1. Ouvrez **votre dépôt GitHub personnel** (créé à partir du template) dans un navigateur
2. Repérez les fichiers ou dossiers suivants (selon votre environnement) :
   - `__pycache__/`
   - `.venv/`
   - `venv/`
   - `*.pyc`
3. Répondez par écrit :
   - À quoi servent ces fichiers ?
   - Pourquoi ces fichiers ne devraient-ils pas être versionnés dans un dépôt Git ?

> **Indice :** ces fichiers sont générés automatiquement par Python et ne contiennent pas de code source que l'on souhaite versionner.

- *Aucune modification n'est demandée à cette étape.*

Étape 2 — Ajouter un `.gitignore`

> **Rappel :** Git ignore uniquement les fichiers non encore suivis. Une fois un fichier commité, `.gitignore` n'a **aucun effet rétroactif**.

Travail demandé

1. À la racine du projet, créez un fichier nommé `.gitignore`
2. Ajoutez au minimum les règles suivantes :

```
__pycache__/
*.pyc
.venv/
venv/
```

> Notes :
>
> - Le dossier `.venv` est utilisé sous Linux/macOS, `venv` est utilisé sous Windows.
> - Les fichiers `__pycache__/` et `*.pyc` sont générés automatiquement par Python.
> - Avec ce fichier, Git ignorera **les nouveaux fichiers** qui correspondent aux règles.

3. Observez :
   - les fichiers sont-ils toujours visibles sur GitHub ?
   - apparaissent-ils encore dans VS Code ?
     - *dans l'explorateur de fichiers*
     - *ou dans l'onglet Source Control*

Question

> Expliquez pourquoi les fichiers sont toujours présents sur GitHub. *Indice : Git suit les fichiers dès qu'ils ont été commitées, `.gitignore` ne peut pas les "oublier" automatiquement.*

Étape 3 — Corriger correctement le dépôt

Travail demandé

Les fichiers doivent :

- rester présents sur votre machine
- disparaître du dépôt GitHub

Méthode 1 — Avec VS Code (recommandée)

1. Onglet *Source Control*
2. Clic droit sur les fichiers/dossiers indésirables
3. Choisir **Remove from Index** (ou * **Remove from Git**, **Untrack file** selon les versions de VS Code / GitLens)
4. Vérifier qu'ils ne sont plus suivis *Vous pouvez vérifier qu'un fichier n'est plus suivi si son icône passe de "M (modifié)"" à un symbole vide ou "U (untracked)" dans Source Control.*
5. Committez avec le message :

   > `Add .gitignore and remove ignored files`

6. Push vers GitHub

Méthode 2 — Avec le terminal (alternative)

```
# Linux / macOS / Git Bash
git rm -r --cached __pycache__
git rm -r --cached .venv
git rm --cached *.pyc
git add .gitignore
git commit -m "Add .gitignore and remove ignored files"
git push
```

```
# Windows PowerShell
git rm -r --cached __pycache__
git rm -r --cached .venv
Get-ChildItem -Recurse -Include *.pyc | ForEach-Object { git rm --cached
$_.FullName }
git add .gitignore
git commit -m "Add .gitignore and remove ignored files"
git push
```

> L'option `--cached` supprime le fichier du suivi Git mais **le laisse sur votre disque**. Sans `--cached`, le fichier serait supprimé du disque.

Étape 4 — Vérification finale

Résultat attendu

- Le dépôt GitHub :
  - contient `.gitignore`
  - ne contient plus `__pycache__`, `.venv`, etc.
- Les fichiers existent toujours en local
- L'historique montre **au moins deux commits**
- Dans VS Code, l'onglet Source Control ne liste plus les fichiers ignorés

Questions

1. À quel moment faut-il créer un `.gitignore` ?
2. Que fait exactement l'option `--cached` ?
3. Que se passerait-il sans `--cached` ?

Rendu attendu

- Lien vers le dépôt GitHub
- Réponses aux questions (quelques lignes suffisent)

Remarques

- **".gitignore ne nettoie pas le passé."**
- Git suit des fichiers explicitement ajoutés, pas des dossiers "par magie".
- Corriger un dépôt est normal, même pour les pros
- Toujours vérifier que les fichiers sont **bien untracked** avant de committer

Erreurs à ne pas faire

- Supprimer les fichiers du disque
- Oublier `--cached`
- Recréer le dépôt depuis zéro

TP Avancé — Nettoyer l'historique Git

> **Attention** Cette activité modifie l'historique Git. À ne faire **que sur un dépôt personnel** ou de TP.

Pré-requis

Ce TP suppose que vous comprenez :

- commit

- historique Git
- dépôt distant

Objectifs

- expliquer pourquoi un `.gitignore` ne suffit pas toujours
- comprendre ce qu'est une réécriture d'historique
- supprimer définitivement des fichiers d'un dépôt Git
- utiliser un outil spécialisé (`git filter-repo`)

Contexte

Dans le TP précédent :

- des fichiers indésirables (`__pycache__`, `.venv`, etc.)
- ont été commités puis supprimés du suivi Git

- Problème : ils **existent toujours dans l'historique** du dépôt.

Étape 1 — Constat

Travail demandé

1. Sur GitHub, ouvrez l'historique du dépôt
2. Naviguez dans un ancien commit
3. Vérifiez que les fichiers indésirables sont toujours visibles

Question

> Pourquoi ces fichiers sont-ils encore accessibles ?

Étape 2 — Comprendre le risque

Cas réels :

- mot de passe commité par erreur
- clé API publiée
- données sensibles

> `.gitignore` **ne protège pas le passé**

Étape 3 — Préparer le nettoyage

Travail demandé

1. Vérifiez que le dépôt est propre :

```
git status
```

2. Créez une sauvegarde (fortement recommandé) :

```
git clone <url-du-repo> backup-repo
```

Étape 4 — Nettoyer l'historique (outil moderne)

Outil utilisé

`git filter-repo`

Installation (selon système)

```
pip install git-filter-repo
```

ou via le gestionnaire de paquets.

Suppression définitive des fichiers

```
git filter-repo \
  --path __pycache__ \
  --path .venv \
  --invert-paths
```

Effet :

- les fichiers sont supprimés de **tous les commits**
- l'historique est réécrit

Étape 5 — Forcer la mise à jour sur GitHub

```
git push --force
```

- `--force` écrase l'historique distant
- acceptable ici car dépôt de TP personnel

Étape 6 — Vérification finale

Résultat attendu

- Les fichiers indésirables :
    - absents du dépôt
    - absents de l'historique

- `.gitignore` toujours présent
- Dépôt fonctionnel

Questions

1. Pourquoi cette opération est-elle dangereuse sur un projet collaboratif ?
2. Quelle alternative existe sans réécriture d'historique ?
3. Dans quel cas est-ce **obligatoire** ?

Remarques

> **"Git se souvient de tout… sauf quand on lui demande explicitement d'oublier."**

- Réécrire l'historique est **exceptionnel**
- Toujours prévenir les collaborateurs
- `.gitignore` doit être créé **au début**

> *"On ne nettoie pas l'historique, on recrée le dépôt proprement."*
>
> - Nouveau dépôt GitHub
> - Nouveau commit propre

À éviter absolument

- `git filter-branch`
- Nettoyage sur un dépôt partagé
- Forcer sans comprendre
