

Le versionning

Dans ce cours nous allons aborder un outil de développement web pour gérer vos projets web complexe et de gérer également l'historique de vos fichiers.

En effet, dans certains projets web complexe, vous allez avoir du mal à visualiser les modifications de vos fichiers et de gérer le déploiement de votre projet.

Nous allons donc utiliser **Git** et l'outil **Github**.

Cette solution va vous permettre de suivre plus facilement l'historique de votre projet et de mieux l'organiser.

C'est un outil essentiel que ce soit pour vos projets seul ou en équipe.

Dans ce cours, nous allons installer et configurer Git sur votre environnement et Github qui vous permettra de gérer votre code.

Le gestionnaire de versions :

Pour comprendre le réel intérêt d'un gestionnaire de version, imaginez que vous avez travaillé pendant une semaine sur un projet web, vous venez de le finaliser et il est fonctionnel. C'est donc la **V1 de votre projet**.

Maintenant, un collègue vous fait des suggestion d'amélioration du projet, vous allez donc créer une **deuxième version de votre projet**, c'est la **V2**. Mais vous avez tout de même besoin de garder un historique de votre travail (V1), afin d'éviter les problèmes (bugs, perte de travail etc...).

C'est là que vous allez avoir besoin d'un **gestionnaire de versions**.

Qu'est-ce qu'un gestionnaire de versions

Un gestionnaire de versions est un programme qui permet aux développeurs de **conserver un historique des modifications et des versions de tous leurs fichiers**.

Le fait de contrôler les versions est aussi appelée "**versioning**" en anglais, vous pourrez entendre les deux termes.

Le gestionnaire de versions permet de garder en mémoire :

- chaque modification de chaque fichier,
- pourquoi elle a eu lieu,
- et qui a fait la modification

Si vous travaillez seul, vous pourrez garder l'historique de vos modifications ou revenir à une version précédente facilement.

Si vous travaillez en équipe, vous saurez qui a modifié quoi plus facilement et vous pourrez éviter des erreur d'écrasement de données si plusieurs personnes travail sur le même projet, ou sur le même fichier.

Cet outil a donc trois grandes fonctionnalités :

1. Revenir à une version précédente de votre code en cas de problème.
2. Suivre l'évolution de votre code étape par étape.
3. Travailler à plusieurs sans risquer de supprimer les modifications des autres collaborateurs.

Pour ce cours, nous allons utiliser le gestionnaire de versions **Git** qui est le plus largement utilisé dans le monde du développement.

Pourquoi est-ce utile dans le travail d'équipe ?

Prenons un exemple concret !

Paul et Jacques travaillent sur le même projet pour un client depuis un mois.

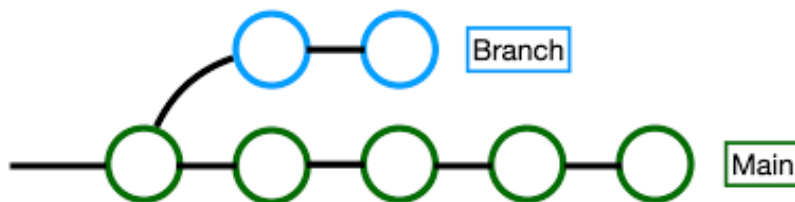
Un jour, le client leur demande de livrer le projet en urgence. Paul finalise donc une dernière modification sur le projet et enregistre son travail.

Mais en faisant ça, il vient accidentellement d'écraser tout le travail de Jacques depuis un mois.

Ils se retrouvent donc avec un projet non fonctionnel et incomplet étant donné que Jacques n'a pas d'historique de son travail et ne peut plus le récupérer.

S'ils avaient travaillé sur un gestionnaire de versions, ils auraient pu très facilement retrouver l'historique et récupérer les modifications de Jacques. Il auraient même pu éviter l'écrasement des données de Jacques car le gestionnaire de version aurait fusionné les deux versions pour éviter d'en perdre une des deux.

De plus, s'ils avaient utilisé Git, ils auraient pu mettre à jour le projet n'importe quand en envoyant ou récupérant les modifications faites par l'un ou l'autre.



Si Paul et Jacques avaient utilisés Git pour leur projet, ils auraient pu travailler chacun sur leur partie du code. Quand ils les regroupent, leurs versions précédentes sont archivées.

Git et Github

Git et GitHub sont deux choses très différentes.

Git est un gestionnaire de versions. Vous l'utiliserez pour créer un dépôt local et gérer les versions de vos fichiers.

GitHub est un service en ligne qui va héberger votre dépôt. Dans ce cas, on parle de **dépôt distant** puisqu'il n'est pas stocké sur votre machine.

C'est un peu confus ? Pas de panique, prenons un exemple pour illustrer cela.

Imaginez, vous participez à la préparation d'un parfum. Chez vous, vous créez la base du parfum en mélangeant différents ingrédients. Puis vous envoyez cette base dans un entrepôt où il sera stocké. Cette base de parfum pourra être distribuée telle quelle ou être modifiée en y ajoutant d'autres ingrédients.

Eh bien, c'est la même chose pour Git et GitHub. Git est la préparation que vous avez réalisée chez vous, et GitHub est l'entrepôt où elle peut être modifiée par les autres ou distribuée.

Le dépôt distant et dépôt local :

Le dépôt local

Un **dépôt local** est un entrepôt virtuel de votre projet. Il vous permet d'enregistrer les versions de votre code et d'y accéder au besoin. Cet entrepôt virtuel est stocké sur votre ordinateur.

Pour illustrer cette idée, prenons l'image de la réalisation d'un parfum. Pour faire un parfum, vous allez réaliser les étapes suivantes :

- Récupérer les fragrances (senteurs) ;
- Les stocker dans un placard pour pouvoir les assembler par la suite ;
- faire le bon mélange ;
- stocker le parfum finalisé ;

Dans cet exemple, le placard est comme un dépôt local : c'est l'endroit où vous stockez vos fragrances au fur et à mesure.

Un **dépôt local** est utilisé de la même manière ! On réalise une version, que l'on va petit à petit améliorer. Ces versions sont stockées au fur et à mesure dans le dépôt local.

Le dépôt distant

Le **dépôt distant** est un peu différent. Il permet de stocker les différentes versions de votre code afin de garder un **historique délocalisé**, c'est à dire que cette fois, vous pouvez stocker votre dépôt sur internet (sur un hébergement ou un serveur).

Imaginez que votre PC ne fonctionne plus et que vous n'avez plus accès à vos fichiers et dossier sur votre ordinateur, si vous faite un dépôt distant, vous pourrez donc le récupérer depuis n'importe quel autre ordinateur.

C'est pourquoi il est important de commencer par copier vos sources sur un dépôt distant, pour garder une sauvegarde sur un hébergement distant. Github est l'un des hébergeurs les plus connus et les plus utilisé.

Le dépôt distant est donc un type de dépôt indispensable si vous travaillez à plusieurs sur le même projet, puisqu'il permet de centraliser le travail de chaque développeur. C'est aussi sur le dépôt distant que toutes les modifications de tous les collaborateurs seront fusionnées.

Alors, pourquoi créer une copie locale ?

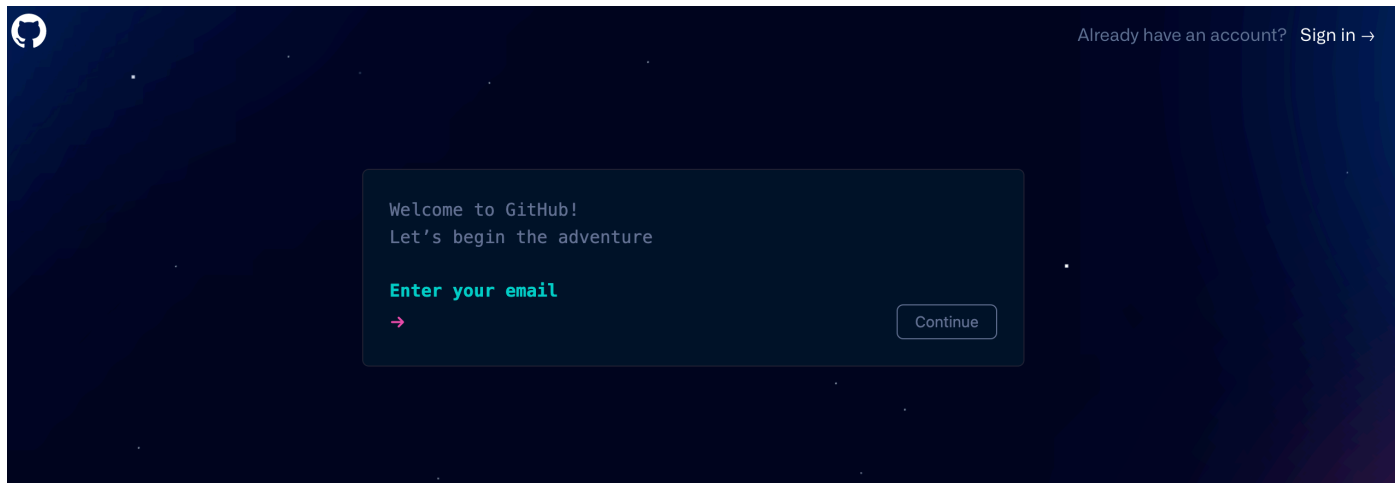
Tout simplement car votre dépôt local est un clone de votre dépôt distant. C'est sur votre dépôt local que vous ferez toutes vos modifications de code.

Ainsi, les dépôts sont utiles si :

- vous souhaitez conserver un historique de votre projet,
- vous travaillez à plusieurs,
- vous souhaitez collaborer à des projets open source,
- vous devez retrouver par qui a été faite chaque modification,
- vous voulez savoir pourquoi chaque modification a eu lieu.

Créer un compte Github :

Rendez-vous sur [Github](https://github.com) pour vous créer un compte :



Une fois que vous avez créé votre compte, nous allons maintenant créer un dépôt distant afin de pouvoir développer la suite du cours.

Configuration de Git

Installation

Maintenant que vous avez créé votre dépôt distant et créé votre compte Github, nous allons configurer votre environnement en installant **Git**, qui va vous permettre de travailler en local (sur votre poste) et de pouvoir ensuite envoyer vos modifications sur votre dépôt distant.

Dans un premier temps, vous devez installer Git :

- Pour les environnement Macbook, bonne nouvelle, vous n'avez simplement qu'à installer Xcode qui est un environnement de développement avancé pour mac qui contient Git.

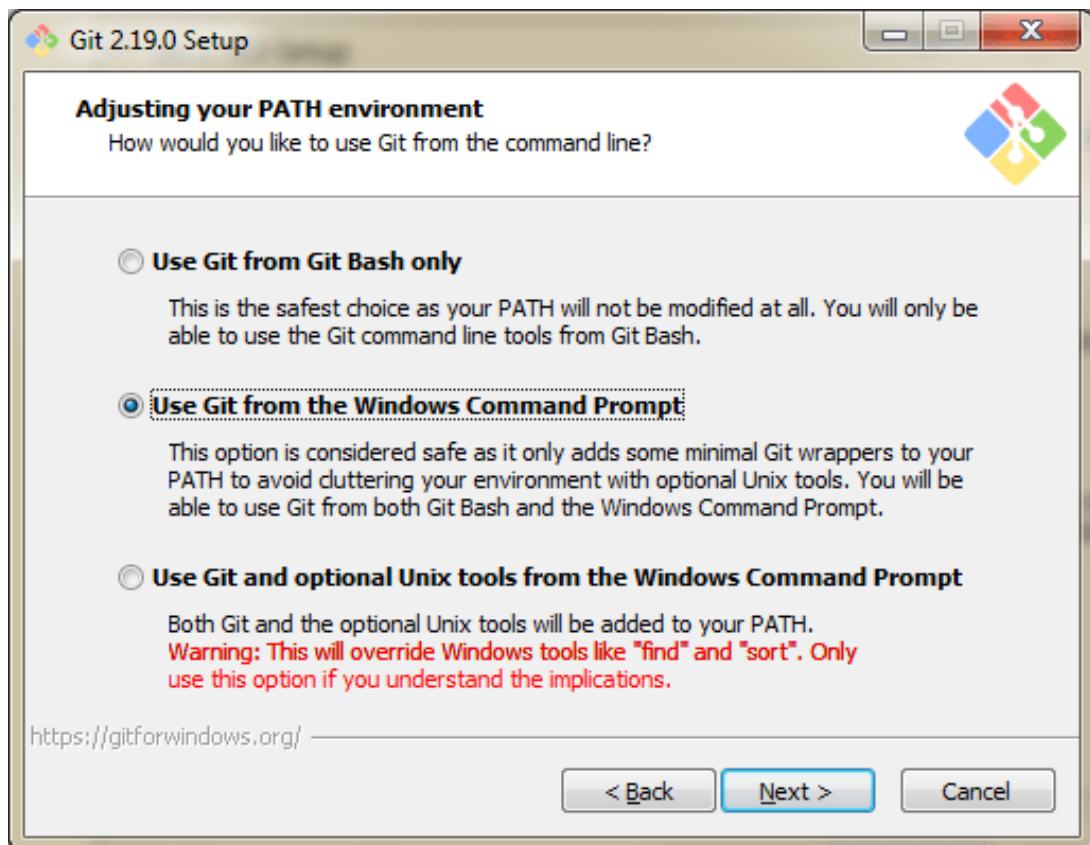
Vous devez donc ouvrir un terminal de commande et entrer la commande

```
xcode-select --install
```

Une fenêtre va s'ouvrir en vous demandant d'installer un programme en ligne de commande, cliquer sur installer.

- Pour les environnement Windows, vous allez devoir vous rendre sur le [site de Git](https://git-scm.com/)
Télécharger le programme d'installation et lancez le.

Dans l'interface d'installation cliquez sur continuer en laissant les paramètres par défaut jusqu'à la fenêtre d'ajustement du PATH, vous devez sélectionner `use Git from the Windows Command Prompt` pour vous permettre d'utiliser Git sur n'importe quel terminal de votre environnement.



Vous avez maintenant installé Git sur votre environnement, il faut le configurer.

Configuration

La prochaine étape va être de faire la configuration de Git sur votre environnement, l'idée est d'avoir le même username et le même email sur Github et sur Git de votre environnement.

Pour faire cette configuration, vous devez ouvrir un terminal de commande et entrer :

```
git config --global user.name "Votre Nom"
```

```
git config --global user.email votreEmail@email.com
```

Maintenant, vous pouvez également configurer les couleurs pour une meilleure lisibilité en ligne de commande (facultatif), vous devez rentrer les commandes suivantes une par une dans le terminal :

```
git config --global color.diff auto
git config --global color.status auto
git config --global color.branch auto
```

Dernière chose à faire : **configurer votre clé SSH.**

Cette clé va permettre d'identifier votre poste (environnement) sur Github et de pouvoir modifier et envoyer des modifications de manière plus simple et plus sécurisé.

Ouvrez un terminal de commande et entrez la commande :

```
ssh-keygen -o

Generating public/private rsa key pair.
Enter file in which to save the key (/user/pierre/.ssh/id_rsa):
Created directory '/user/pierre/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /user/pierre/.ssh/id_rsa.
Your public key has been saved in /user/pierre/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

Une fois que vous avez terminé cette étape, vous n'avez plus qu'à récupérer la clé SSH et de la définir dans Github. Pour récupérer la clé, vous devez entrer la commande :

```
cat ~/.ssh/id_rsa.pub
```

Copiez toute la clé qui commence normalement par `ssh-rsa` et rendez vous sur Github :

- Cliquez sur votre icône en haut à droite et sélectionnez dans le menu déroulant `Settings`,
- Ensuite, cherchez l'onglet `SSH and GPG keys`
- Cliquez sur `new SSH key`
- Collez votre clé dans le block `key` et donnez lui un titre (celui que vous voulez)
- Cliquez sur Add Key.

Votre clé SSH est maintenant identifiée sur Github et vous pouvez maintenant gérer les interactions entre Github et votre environnement plus simplement.

Initialiser un dépôt local

Maintenant que nous avons finalisé la configuration de Git et Github, nous allons pouvoir faire notre premier dépôt !

Nous allons maintenant initialiser un dépôt local (donc sur votre ordinateur).

Pour ça, vous allez devoir créer un dossier sur votre ordinateur "TestGit" par exemple.

Maintenant, c'est dans le terminal de commande que tout se joue.

Vous devez ouvrir un terminal de commande et aller dans le dossier que vous venez de créer en ligne de commande.

Pour naviguer dans vos dossiers en ligne de commande vous devez connaître certaines commandes utiles

- `Ls` (windows : `dir`) -> va afficher la liste de tous les fichiers et dossiers du répertoire actuel

```
ls ou dir
```

- **cd nomDuDossier** -> cd pour change directory, en utilisant la commande cd + le nom de votre dossier, vous changez le répertoire de destination du terminal, autrement dit, vous naviguez dans le dossier avec le terminal

```
cd home
```

- **cd ..** -> Cette commande va également changer de répertoire (dossier), mais cette fois va revenir dans le répertoire parent

```
cd ..
```

Vous devez donc vous rendre dans le dossier que vous avez créé qui est pour le moment vide pour initialiser le dépôt.

Une fois que vous êtes bien dans votre dossier avec le terminal de commande, il ne vous reste plus qu'une commande à lancer pour initialiser votre dépôt : **git init**

```
git init
```

Le terminal va vous renvoyer un message `Initialized empty Git repository in...`, cela veut dire que Git vient d'initialiser un dépôt local avec votre dossier.

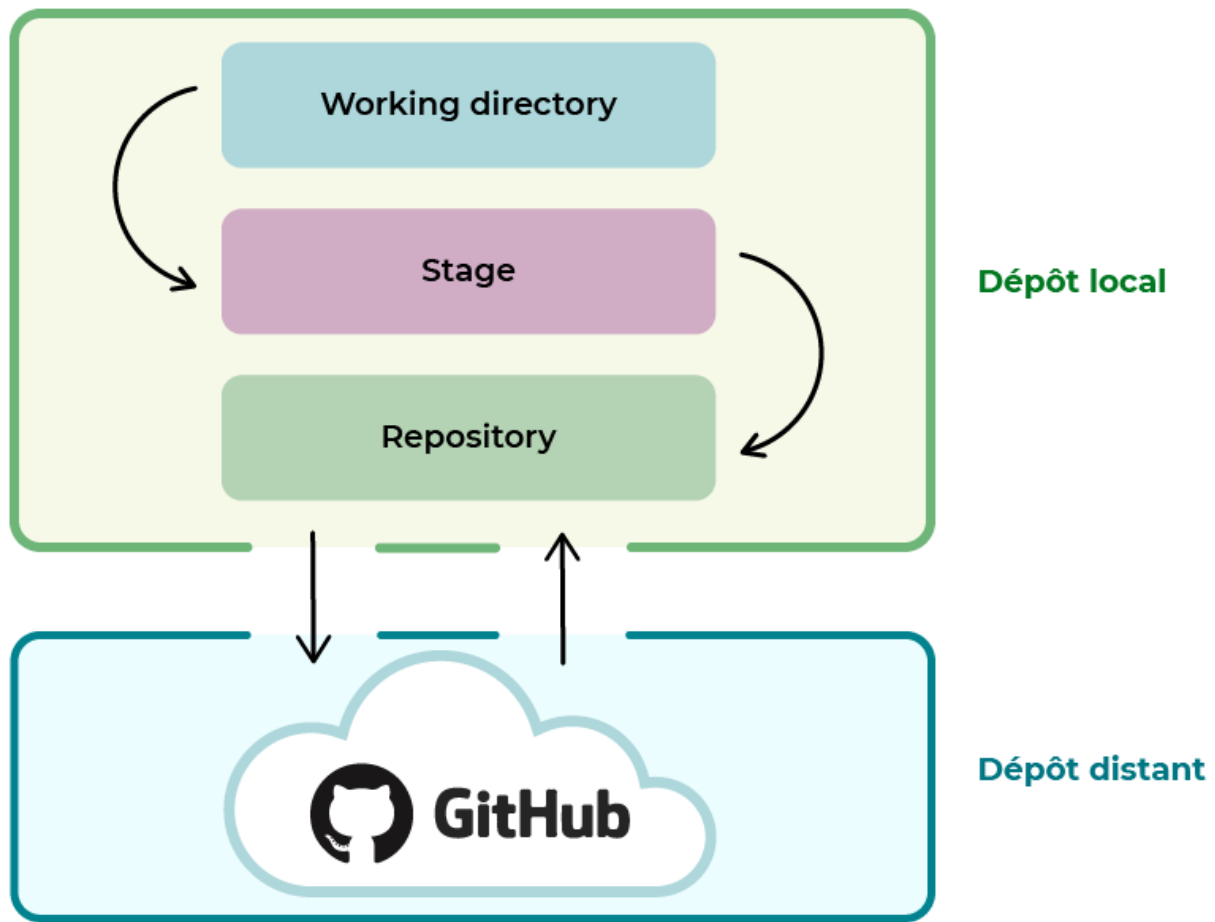
Si vous retournez dans l'explorateur de fichier, vous trouverez à l'intérieur un dossier caché `.git` qui va contenir les logs de Git, la configuration de votre dépôt, les branches etc...

Modifier son dépôt :

Maintenant que nous avons initialisé notre dépôt, nous allons pouvoir travailler dessus et faire nos premières modifications.

Mais avant toute chose, il faut comprendre comment Git fonctionne pour versionner notre projet.

Prenons le temps d'analyser ce schéma :



Dans un premier temps nous avons le **Working directory** c'est notre dossier de travail sur votre ordinateur (le fameux dossier TestGit).

Ensuite nous avons la partie **Stage** aussi appelé **Index**, cette zone va accueillir tous les fichiers que vous allez modifier et que vous voulez voir apparaître dans votre prochain version du projet.

Et enfin, nous avons le **Repository**, c'est dans cette zone que les différentes versions de votre projet seront stockées (le fameux placard où vous placez vos fragrances pour la fabrication du parfum).

C'est 3 zones sont présente dans votre dépôt local (donc sur votre ordinateur).

Enfin, nous voyons sur le schéma la zone Github, c'est tout simplement votre dépôt distant (votre projet qui est stocké sur Github).

Pour un exemple plus concret :

Imaginons que nous avons un projet qui est composé de 3 fichiers. Ce projet est donc notre dépôt local.

Nous modifions notre premier fichier ainsi que le deuxième depuis notre working directory. Nous avons maintenant une version évoluée de notre projet et nous aimerions sauvegarder cette version grâce à Git, donc de la stocker dans le repository.

Pour ça nous allons commencer par envoyer les fichiers modifiés du working directory vers l'index, on va donc **indexer les fichiers modifiés**, une fois que les fichiers seront indexés, nous pourrons créer une nouvelle version de notre projet.

Cas pratique

Maintenant que nous avons vu la théorie, nous allons pouvoir passer à la pratique, nous allons modifier notre dépôt local "TestGit" et versionner notre projet.

Vous devez en amont avoir initialisé le dépôt avec la commande `git init` dans votre projet.

Ouvrez votre dossier TestGit avec l'IDE **Vscode**, il va nous être utile pour faire les modifications de notre dépôt et ensuite entrer les commandes dans le terminal intégré de VsCode.

Une fois votre dossier ouvert, vous allez créer un nouveau fichier, pour ça 2 solutions :

- **Avec VsCode** dans la partie de gauche (architecture dossier et fichier), faites un clic droit > `nouveau fichier`, écrivez le nom : **algo.py** et validez avec entrée.
- **En ligne de commande** : Ouvrez un terminal VsCode (menu Vscode > terminal > nouveau terminal). Un terminal va s'ouvrir avec le bon emplacement (directement dans l'emplacement de votre projet). Entrez ensuite la commande de création de fichier :

```
touch algo.py
```

La commande touch veut dire en ligne de commande -> création de fichier.

Vous venez de créer un premier fichier python dans votre projet

Maintenant, si vous regardez bien dans la liste de fichier dans VsCode (onglet de gauche), vous voyez bien votre nouveau fichier en vert avec un petit `U` à côté, VsCode a compris que vous avez modifié le working directory, et sait que vous venez d'ajouter un fichier par rapport à la première version de votre projet.

Pour le moment, nous n'avons pas indexé cette modification, elle est seulement sur le working directory, mais nous n'avons pas archivé cette nouvelle version.

Indexer la modification

Nous voulons maintenant ajouter la modification du projet à l'index pour pouvoir créer une deuxième version de notre projet.

Dans un premier temps, vous pouvez afficher les fichiers qui ont eu une modification, mais qui ne sont pas encore indexés grâce à la commande :



```
git status
```

Pour indexer des modifications, vous devez connaître une commande :

```
git add votreFichier
```

Dans notre cas, nous devons ouvrir un terminal VsCode, et rentrer la commande :

```
git add algo.py
```

En rentrant cette commande, vous verrez qu'à côté du fichier, l'icone  a été remplacé par l'icone  ce qui veut dire que le fichier a été indexé, mais pour le moment, nous n'avons pas encore créé de nouvelle version du projet.

Si vous avez plusieurs fichiers à indexer en même temps, vous pouvez utiliser la commande

```
git add .
```

Le point veut dire "ajoute tous les fichiers" à git.


Versionner le projet

Dernière étape pour finaliser la nouvelle version du projet, l'archivage de la nouvelle version.

Pour ça nous allons faire ce qu'on appelle un **commit** (terme de Git qui représente l'archivage d'une version du projet).

Pour faire ce commit, nous allons rentrer la commande :


```
git commit -m "add Python file"
```

L'argument  permet d'ajouter un titre à votre version, il est très important d'en ajouter pour chaque commit, cela permet de savoir ce que ce commit a modifié sur votre projet.

Créer un dépôt distant sur Github

Maintenant que nous avons fait une première version de notre projet sur notre dépôt local, nous voulons envoyer le projet sur un dépôt distant (sur Github). Nous allons donc créer un repository sur Github.

Un repository dans Github est tout simplement un dépôt hébergé sur Github.

Vous pouvez en créer un en cliquant sur le  à côté de votre icon de profil sur la barre de navigation à droite, et sur `new repository`.

Ensuite, ajouter le nom de votre dépôt, la confidentialité (public ou privé), et **ne cochez pas** add Readme file et add gitignore.

Ces 2 options vous permettent de créer un **readme** (fichier de description du projet) et un **gitignore** qui permet d'ignorer certains fichiers dans le versionning du code.

Ensuite cliquez sur `create repository`


Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?

[Import a repository.](#)

Owner *

Repository name *



 Pierre-brtrd ▾

 /

cours-versionning ✓

Great repository names are short and memorable. Need inspiration? How about [symmetrical-octo-fiesta](#)?

Description (optional)

- ☒  **Public**
Anyone on the internet can see this repository. You choose who can commit.
- ☐  **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

- ☒ **Add a README file**
This is where you can write a long description for your project. [Learn more.](#)
- ☒ **Add .gitignore**
Choose which files not to track from a list of templates. [Learn more.](#)

.gitignore template: None ▾

Félicitation, vous venez de créer votre premier dépôt distant.

Maintenant, vous avez une fenêtre qui vous donne les commandes à entrer dans un terminal pour connecter votre dépôt distant à un dépôt local :

Quick setup — if you've done this kind of thing before

 Set up in Desktop or ☐ HTTPS ☐ SSH

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# Formation_Git" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M master
git remote add origin git@github.com:Pierre-brtrd/Formation_Git.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin git@github.com:Pierre-brtrd/Formation_Git.git
git branch -M master
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#)

Intéressons nous à la partie `...or push an existing repository from the command line`, Github vous donne les 3 commandes à rentrer pour connecter votre dépôt local à votre nouveau dépôt distant.

Dans un premier temps (sur le dépôt local) :

```
git remote add origin urlReposDistant
```

Cette commande va ajouter une connexion entre les deux dépôt (local et distant).

Copier cette première ligne (avec l'url de votre dépôt distant) et copiez la dans le terminal VsCode.

Vous venez de créer la connexion entre vos deux dépôt !

Maintenant, nous devons rentrer la commande :

```
git branch -M master
```

qui va permettre de créer si ce n'est pas déjà fait une branch master (la branch d'origine de vos dépôt).

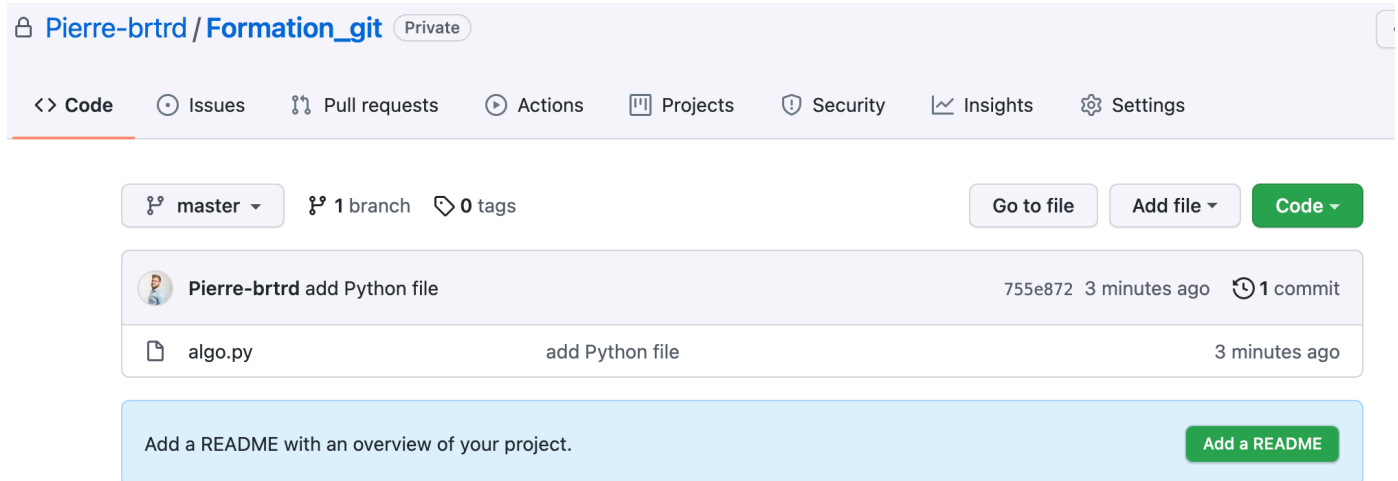
Dernière étape, envoyer le dépôt local, sur votre dépôt distant :

```
git push origin master
```

Cette commande va envoyer votre dépôt local, sur le dépôt distant, grâce à la commande `git push`, ensuite nous avons défini la branch de destination sur le dépôt distant (où nous envoyons cette version du projet), grâce à la commande `origin master` qui spécifie que nous envoyons cette version du code sur le dépôt distant dans la branch `master`.

La branch **master** est dans la plupart des cas, la branch d'origine de votre projet.

Maintenant, si vous allez sur Github et que vous cliquez sur le nom de votre dépôt, vous verrez que votre fichier est bien présent :



Récupérer une modification depuis le dépôt distant

Nous venons de voir comment envoyer des modifications depuis notre dépôt local vers notre dépôt distant.

Maintenant, nous allons faire une modification sur le dépôt distant et la récupérer sur notre dépôt local.

Si vous regardez sur le repos Github, vous verrez dans le bandeau bleu sous vos fichiers le bouton **Add a README**, ce fichier README est le fichier qui va décrire votre projet, il est sur TOUS les repos Github, et permet d'avoir une description du repos.

Nous allons créer ce README directement sur Github, et récupérer la modification (l'ajout du fichier) sur notre dépôt local.

Cliquez sur le bouton README sur Github.

Vous allez alors entrer dans l'éditeur de Github, le fichier README est écrit en Markdown (langage de programmation léger qui permet de mettre en forme du texte)

Ajouter une brève description dans le fichier et terminez par cliquer sur **Commit new file** en bas de la page.

Cette action va automatiquement créer une nouvelle version du repos avec l'ajout de votre nouveau fichier.

Nous avons maintenant une nouvelle version de notre dépôt distant, mais pour le moment, notre dépôt local n'a pas encore eu ces nouvelles modifications.

Nous devons **tirer** les modifications sur le dépôt distant vers notre dépôt local.

Retournez donc sur VsCode et ouvrez un terminal.

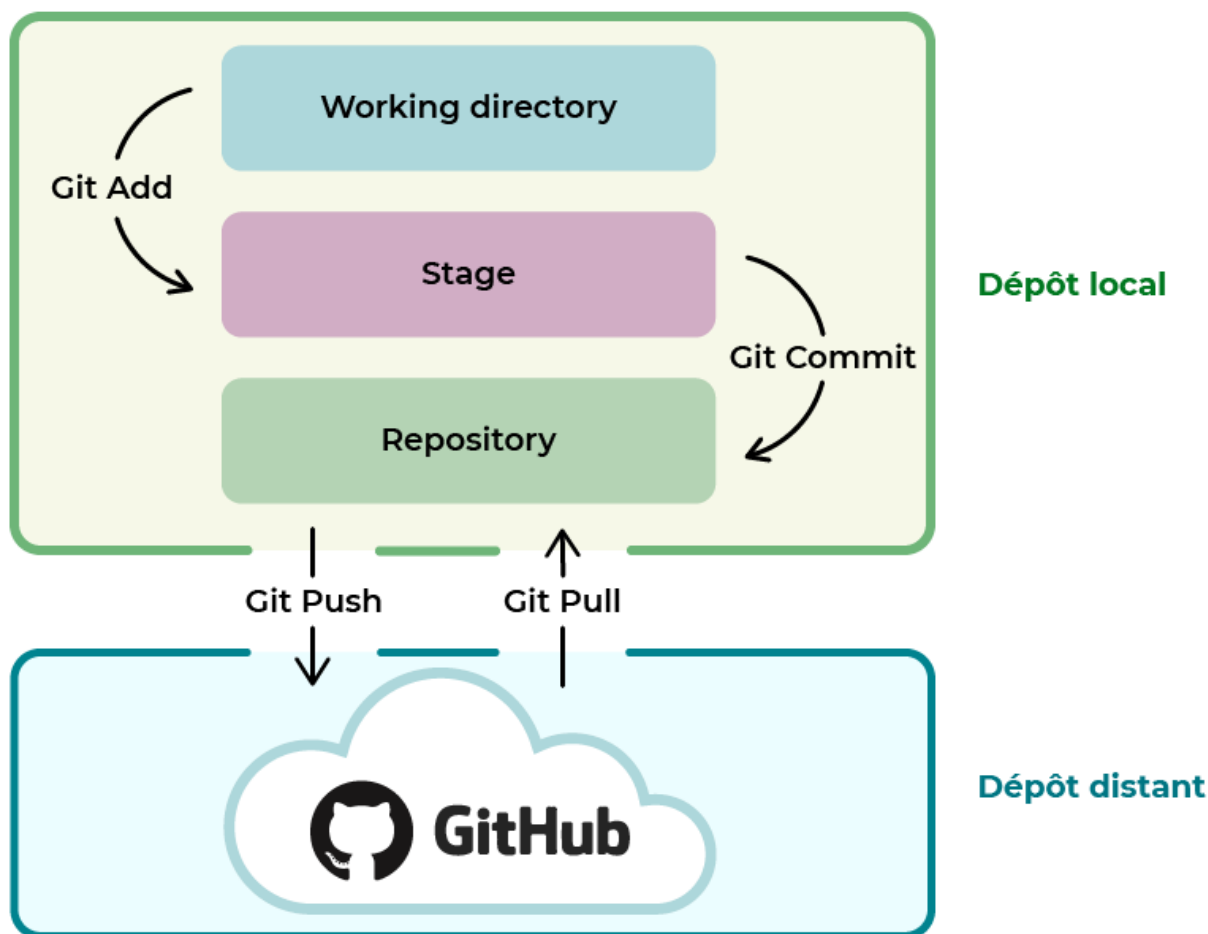
Nous allons utiliser la commande `git pull` -> (pour tirer cette fois)

```
git pull origin master
```

Nous utilisons `git pull` pour dire à git -> tu vas récupérer des informations depuis un dépôt distant, ensuite nous indiquons quelle branch doit être récupérée `origin master`.

Et avec cette commande, vous allez voir que le fichier README que nous avons créé sur Github, arrive bien sur votre dépôt local !

Voici le schéma complet de Git par rapport à ce que nous venons de faire :



Et voilà, vous savez maintenant comment gérer le versionning de votre code, nous allons maintenant voir un concept très important et surtout TRÈS pratique avec cette solution de développement : **LES BRANCHES**.

Les branches

Le principal atout d'utiliser Git pour vos projets repose sur 1 concept : **LES BRANCHES**

Les différentes branches d'un dépôt représente des copies du code principal à un instant T, et c'est grâce à ces branches que vous allez pouvoir tester l'ajout de fonctionnalité à votre projet par exemple, mais sans impacter votre code principal !

Nous avons vu qu'avec Git la branch principale est appelé **master** (ou main) et cette branch portera par la suite l'ensemble des modifications apportées sur votre projet. Mais le but n'est donc pas de réaliser ces modifications directement sur cette branche, mais de réaliser les modifications sur d'autres branches, et après des tests pour vérifier que votre code marche bien, intégrer ces nouvelles modifications sur la branche principale (master).

Prenons un exemple concrêt :

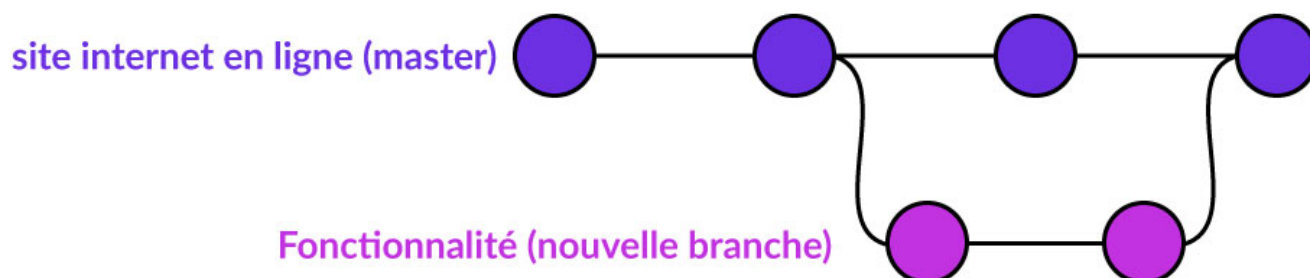
Imaginons que vous venez de terminer un projet de site web, le **site est en ligne** et tout votre code est sur la **branche master (principale)**.

Mais votre client veut maintenant **intégrer l'ajout d'une fonctionnalité d'inscription à une newsletter**.

Si vous faites votre développement sur la branche principale, vous pouvez **potentiellement rendre le site non fonctionnel** si vous faites une erreur dans votre code.

Vous voulez donc développer entièrement la nouvelle fonctionnalité, faire vos tests d'intégration, et seulement **une fois que vous avez vérifié que tout marche bien, intégrer ces nouvelles modifications sur votre site** (donc sur la branche master).

C'est là que vous allez utiliser **une nouvelle branche** ! Vous allez créer une branche à partir de votre branche master (ce qui va créer une copie de la branche master), ensuite vous aller faire vos modifications sur la nouvelle branche (ce qui n'impactera pas pour le moment la branche master), et une fois les vérifications passées, vous allez intégrer la branche de développement à la branche master pour intégrer la nouvelle fonctionnalité sur le site.



Création d'une nouvelle branche sur notre dépôt local

Maintenant, nous allons mettre en pratique cette notion.

Dans notre projet, nous avons seulement créé un fichier python, nous pouvons nous dire que nous voulons ajouter un premier algorithme dans notre fichier.

Pour ça, nous n'allons pas effectuer les modifications sur la branche master, mais nous allons créer une nouvelle branche.

Pour la créer, nous allons rentrer la commande :

```
git branch init-dev-algo
```

Cette commande nous a créé une nouvelle branche du nom de `init-dev-algo`, nous allons maintenant devoir changer de branche (pour le moment, nous sommes toujours sur la branche master !).

Pour afficher toutes les branches d'un dépôt, vous pouvez entrer la commande :

```
git branch
```

Cette commande doit vous renvoyer :

```
init-dev-algo
* master
```

L'étoile devant `master` indique qu'actuellement, nous sommes dans la branche master, nous savons donc qu'il faut changer de branche avant de commencer le développement.

Pour ça, nous allons rentrer la commande :

```
git checkout init-dev-algo
```

Si nous relançons la commande `git branch` elle va maintenant renvoyer :

```
* init-dev-algo
  master
```

L'étoile est maintenant sur la branch `init-dev-algo` nous sommes donc maintenant sur la bonne branche, nous pouvons débiter le développement.

Ajoutez maintenant un algorithme naïf à votre fichier python (block de code sans fonctions)

Enregistrez vos modifications, et pour finir, nous allons indexer la modification du fichier et faire notre commit.

Donc, d'abord :

```
git add algo.py
```

Ensuite :

```
git commit -m "Ajout de la V1 fichier algo"
```

Vous venez de faire un commit sur la branch `init-dev-algo`, maintenant, envoyons ces modifications sur le dépôt distant :

```
git push origin init-dev-algo
```

Cette fois nous spécifions que la branche de destination dans le dépôt distant est `init-dev-algo` car nous voulons créer une nouvelle branche avec nos modifications sur le dépôt distant.

Fusionner les branches

Nous avons donc 2 branches, nous avons vérifié que la branche `init-dev-algo` était fonctionnelle et nous voulons maintenant que ces nouvelles modifications soient intégrées à la branche master.

Nous allons donc **fusionner** les deux branches, on parle de **merger** notre branche de développement sur la branche master.

Vous pouvez le faire directement sur votre dépôt local, et envoyer ensuite cette fusion sur le dépôt distant.

Dans un premier temps, la commande pour effectuer le merge doit se faire TOUJOURS à partir de la branche où nous voulons intégrer les modifications, dans notre cas c'est la branche master.

La première chose à faire est donc de se rendre sur notre branche master :

```
git checkout master
```

Une fois sur la bonne branch, nous allons lancer la commande de merge :

```
git merge init-dev-algo
```

`git merge` sert à dire que nous voulons merger une branche sur notre branche actuelle (donc nous ajoutons le nom de la branche init-dev-algo).

Avec cette commande, vous venez de fusionner vos branches, et la branche master est maintenant bien à jour avec les modifications que vous avez faites sur la branche init-dev-algo.

Envoyez maintenant le merge sur le dépôt distant :

```
git push origin master
```

Si vous retournez sur Github, vous allez voir que la branche master a intégré les modifications (commit) de votre branche init-dev-algo.

Créez votre environnement bac à sable

Nous allons maintenant voir la partie de gestions des éventuelles erreurs lors de votre utilisation de Git et Github.

Pour commencer, vous devez vous créer un nouveau dépôt local et le connecté à un dépôt distant sur Github.

Créez votre dépôt distant, faites un commit pour pouvoir envoyer les modifications dans votre dépôt distant sur Github.

Pour vous aider voici les commandes utiles :

- **Initialiser un dépôt :**

```
git init
```

- **Créer une nouvelle branche :**

```
git branch nom-de-votre-branch
```

- **Naviguer vers une branche :**

```
git checkout nom-de-la-branch
```

- **Voir la liste des fichiers modifiés par encore indexés :**

```
git status
```

- **Ajouter un fichier à l'index :**

```
git add votre-fichier
```

OU pour indexer tous les fichiers modifiés

```
git add .
```

- **Faire le commit :**

```
git commit -m "Le message de votre commit"
```

- **Créer son dépôt distant sur Github et le connecter au dépôt distant :**

Une fois le dépôt distant créé aller dans votre dépôt local et entrez :

```
git remote add origin url-du-dépôt-distant
```

- **Envoyer les modifications vers le dépôt distant :**

```
git push origin nom-de-la-branch
```

- **Merger une branche dans une autre :**

Vous devez être dans la branche de destination (celle qui va intégrer l'autre branche)

```
git merge nom-de-la-branche-a-fusionner
```

Gérer les erreurs d'utilisation de Git :

Maintenant que vous avez les bases, nous allons devoir voir comment pouvoir faire des modifications si vous faites des erreurs en utilisant Git (commits dans la mauvaise branche, suppression d'une branche etc...)

Nous allons utiliser votre environnement Bac à sable pour faire plusieurs tests d'erreur d'utilisation et nous verrons les bonnes pratiques pour les résoudre.

Supprimer une branche

Vous avez créé une branche que vous ne vouliez pas, ou bien que vous avez mal écrite. Vous devez donc la supprimer pour en créer une autre.

Pour effectuer le test, créez une branche :

```
git branch brancheTest
```

Vous venez de créer une branche qui se nomme `brancheTest`.

Maintenant, vous vous rendez compte que vous l'avez mal écrite, donc vous voulez la supprimer.

Dans un premier temps, vérifiez que vous n'êtes pas dans la branche que vous souhaitez supprimer (sinon Git vous renverra une erreur car il ne peut pas supprimer la branche sur laquelle vous êtes).

```
git branch
```

Pour supprimer la branche inutile entrez la commande :

```
git branch -d brancheTest
```

L'argument `-d` veut dire delete (pour supprimer) et ensuite nous indiquons le nom de la branche à supprimer.

Si vous avez déjà fait 1 commit ou plus sur la branche que vous voulez supprimer, vous devrez utiliser une autre commande :

```
git branch -D brancheTest
```

Cette fois, nous passons l'argument `-D` pour supprimer la branche + les commits qui ont été effectués. **ATTENTION**, si vous faites ça, vous supprimez la branche + les modifications que vous avez faites dessus.

Félicitation, vous venez de comprendre comment supprimer une branche sur Git.

Faire une modification sur la mauvaise branche

Ce qui peut arriver, c'est de faire une modification et de faire le commit, mais de se rendre compte après coups que vous n'étiez pas dans la bonne branche !

C'est un cas un peu plus complexe car vous avez enregistré vos modifications, et le commit est déjà fait sur votre branch.

Dans un premier temps, vous allez devoir vous plonger dans le git log (l'historique de vos commits) :

```
git log
```

```
commit 68d5ae6328596b2887e8ffafd33095f42f75361a (HEAD -> brancheTest)
Author: pierre.bertrand.professionnel <pierre.bertrand.professionnel@gmail.com>
Date:   Wed Oct 20 14:23:38 2021 +0200

commit suppr
```

Voici un exemple de commit dans votre git log, nous voulons récupérer seulement le hash, c'est la longue chaîne de caractère qui est intégré juste après commit, dans l'exemple :

```
68d5ae6328596b2887e8ffafd33095f42f75361a.
```

Dans un premier temps, vous allez devoir récupérer ce hash afin de pouvoir supprimer le commit que vous venez de faire sur la branche master, et de déplacer ce commit dans la bonne branche.

Donc tout d'abord, on récupère le hash du log du commit que nous voulons changer de branche.

Ensuite sur la branche master, vous allez devoir rentrer la commande pour supprimer le dernier commit que vous venez de faire :

```
git reset --hard HEAD^
```

le `HEAD^` signifie que vous voulez supprimer le dernier commit de cette branche.

Ensuite nous allons aller sur la branche dans laquelle nous devons faire ces modifications :

```
git checkout brancheTest
```

Une fois dessus, vous allez entrer la commande pour intégrer le dernier commit que nous avons supprimé de la branche master :

```
git reset --hard 68d5ae6328596b2887e8ffafd33095f42f75361a
```

Modifier le nom du dernier commit

Une erreur également fréquente est d'avoir mal écrit le nom de son dernier commit (faute de frappe par exemple).

Pour ça, il existe une commande assez simple qui permet de réécrire le message de votre dernier commit :

```
git commit --amend -m "Nouveau message"
```

L'argument `--amend` avant l'argument `-m` vous permet de dire à Git que vous voulez modifier le message du dernier commit.

Ajouter un fichier au dernier commit

Dans certains cas, vous allez faire votre commit, mais vous vous rendez compte après qu'il vous manque une modification de fichiers, plutôt que de devoir faire un autre commit, vous pouvez ajouter votre nouveau fichier à l'index `git add .` Et ensuite faire le commit avec simplement l'argument `--amend`:

```
git commit --amend --no-edit
```

Nous rajoutons seulement l'argument `--no-edit` pour dire que nous ne souhaitons pas modifier le commit (son nom) mais que nous voulons simplement ajouter les fichiers dans l'index au dernier commit.

La gestion des erreurs sur Github

Nous avons vu les bonnes pratiques pour modifier des erreurs en locale, mais le problème est différent si vous avez déjà effectué les modifications sur le dépôt distant.

Github va garder un historique de toute vos modifications, donc, si vous avez pusher des modifications en erreurs, vous ne pourrais pas facilement supprimer ce commit (c'est le principe de Github de garder un historique de toutes vos modifications) il est donc difficile de supprimer une partie de l'historique sur un dépôt distant.

Mais il reste tout de même une solution si vous avez besoin d'annuler les modifications que vous avez push : le `git revert`

```
git revert HEAD^
```

Mais cette fois, le `git revert` va refaire un nouveau commit en supprimant les modifications que contient le dernier commit.

Ce qui fait que vous avez toujours dans votre historique les 2 commits (celui avec les mauvaises modifications, et l'autre qui les supprime).

Revenir à une version précédente

Imaginons que votre client vous demande de travailler sur une nouvelle fonctionnalité sur un projet.

Vous faites votre développement et le lendemain, le client change d'avis, mais vous avez déjà effectué des modifications et vous voulez donc revenir à une version précédente de votre projet.

C'est possible avec Git !

Vous pouvez utiliser la commande `git reset` qui permet de revenir à une ancienne version de votre dépôt.

```
git reset NomDuCommit --hard
```

Pour la commande `git reset` nous devons lui indiquer la cible (la version que nous voulons réinitialiser), on lui passe également l'argument `--hard` pour effectuer la réinitialisation complète

(SOYER SÛR DE VOUS EN FAISANT CETTE COMMANDE : TOUS LES FICHIERS SERONT RÉINITIALISÉS !)

Ce qui veut dire que nous allons revenir sur une ancienne version mais en oubliant tout les fichiers que vous avez modifiés après ce commit.

Donc, si vous voulez revenir à une ancienne version du projet vous devez :

- Récupérer le hash du commit cible :

```
git log
```

```
commit 82a01f5183c6fcdca247bc98a3fbdb13336e4192 (master)
Author: pierre.bertrand.professionnel <pierre.bertrand.professionnel@gmail.com>
Date:   Mon Oct 25 12:47:09 2021 +0200

    Add premierFichier File
```

Ici la longue chaîne de caractère sur la première ligne.

- Rentrer la commande de reset :

```
git reset --hard 82a01f5183c6fcdca247bc98a3fbdb13336e4192
```

Maintenant, dans le cas où vous souhaitez revenir à une ancienne version mais ne pas perdre les modifications vous pouvez effectuer la même commande, mais en passant cette fois l'argument `--mixed` qui va permettre de revenir à une version antérieure, mais de ne pas perdre les modifications (elle seront dans votre working directory et vous pourrez les ajouter si besoin).

Donc la commande pour ne pas perdre les modifications en revenant à une ancienne version est :

```
git reset --mixed 82a01f5183c6fcdca247bc98a3fbdb13336e4192
```

Gérer les conflits

Nous avons vu comment fusionner des branches (merger). Mais il se peut que tout ne se passe pas comme prévu dans certains cas.

En utilisant Git et Github, vous verrez qu'il peut arriver que nous ayons des conflits.

Un conflit c'est quand vous voulez fusionner des branches ensemble, mais que les branches ont des modifications sur le même fichier ET sur la même ligne.

Prenons un exemple : vous avez modifié sur la branche master le fichier algo.py et aussi sur une autre branche de développement.

Vous voulez maintenant fusionner votre branche de développement, mais cette fois, pendant la fusion git va avoir un problème : les 2 branches font chacune une modification sur la même ligne du même fichier et ne sait pas quelle modification doit être prise en compte, c'est un conflit !

Tout d'abord ouvrez une nouvelle branche à partir de master, ensuite modifier votre fichier algo.py sur la branche master, faites un commit, et ensuite modifier la même ligne du même fichier sur l'autre branche.

Maintenant retournez sur la branche master pour effectuer le merge de votre nouvelle branche :

```
git checkout master
```

```
git merge modif-algo
```

Maintenant, VsCode devrait vous répondre une erreur : Il y a un conflit !

```
<<<<<<< HEAD
    nombre1 = recup_valeur1()
=====
    nombre1 = recup_valeur1_branche()
>>>>>>> cours-conflit
```

Ici le **HEAD** est la modification que vous avez faite sur la branche actuelle (master).

cours-conflit est la modification faite sur l'autre branche.

Pour pouvoir finaliser le merge des deux branches, vous devez choisir une des deux modifications, pour ça, vous avez simplement à supprimer la modification non voulu.

Imaginons que nous voulons garder ce que nous avons écrit sur master, nous devons donc supprimer la modification **cours-conflit**, alors nous avons simplement à supprimer :

```
=====
    nombre1 = recup_valeur1_branche()
>>>>>>> cours-conflit
```

Une fois que c'est fait, vous pouvez supprimer le `<<<<<<< HEAD` pour que le fichier ne comporte plus de conflit et retrouver un code propre et sans indication de conflit git.

Vous pouvez maintenant enregistrer les modifications et faire le commit :

```
git add .
```

```
git commit -m "résolution de conflit"
```

Félicitation ! Vous avez maintenant toutes les bases pour utiliser Github